Reductie van het geheugengebruik van besturingssysteemkernen

Memory Footprint Reduction for Operating System Kernels

Dominique Chanet

Promotor: prof. dr. ir. K. De Bosschere
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2006 - 2007

UNIVERSITEIT
GENT

ISBN 978-90-8578-168-4
NUR 980
Wettelijk depot: D/2007/10.500/42

Voor An,
mijn vrouw, maar vooral
mijn steun en toeverlaat

Voor Jozef Ghijsens,
die wel het begin maar niet meer het einde
van deze reis mocht meemaken

Woord vooraf

Het werk dat nu voor u ligt, zou niet tot stand gekomen zijn zonder de hulp, raad en steun van heel wat mensen. Ik wil deze dan ook graag bedanken in dit woord vooraf.

Allereerst is er natuurlijk mijn promotor, Prof. De Bosschere, die me de kans gaf om dit onderzoek uit te voeren, en die steeds klaar stond met raad of een bemoedigend woord als dat nodig was. Ook Bjorn De Sutter en Bruno De Bus, collega's en mentors, verdienen een speciale vermelding. Je kan wel stellen dat ik van hen het vak geleerd heb. Behalve Bjorn en Bruno waren ook Ludo, Matias en Bertrand medereizigers in het grote Diablo-avontuur. Ze waren niet alleen fijne collega's, maar stonden ook garant voor de nodige ontspanning op tijd en stond. Nu mijn onderzoek, en dat van hen, afgerond is, en onze wegen scheiden, kan ik zonder twijfel zeggen dat ik hen allemaal zal missen.

Ook de leden van mijn doctoraatsjury wil ik bedanken voor de tijd en moeite die ze besteed hebben aan het nalezen van deze doctoraatsverhandeling, en de suggesties die ze gedaan hebben om de tekst duidelijker en correcter te maken.

Ik bedank ook graag het Fonds voor Wetenschappelijk Onderzoek Vlaanderen, dat me vier jaar lang van de nodige middelen voorzag om het gevoerde onderzoek te financieren.

Mijn ouders dank ik voor de levenslange steun en aanmoediging. Van jongs af aan hebben ze mijn leergierigheid en interesse in de wereld gestimuleerd, en zonder die impulsen zou ik nooit dit levenspad bewandeld hebben.

Het uitvoeren van een doctoraatsonderzoek en het schrijven van een doctoraatsverhandeling zijn taken die niet stoppen als je 's avonds het kantoor verlaat en naar huis gaat. An, mijn vrouw, had steeds geduld met me als ik 's avonds weer eens wou doorwerken, of als ik er niet met mijn gedachten bij was omdat ik nog over een probleem aan

het nadenken was. Zij gaf me steun als ik die nodig had, en bleef in me geloven, ook op de momenten dat ik dat zelf even niet meer deed. Ik draag dit proefschrift dan ook met liefde aan haar op.

Dominique Chanet
Gent, juni 2007

Samenvatting

In ingebedde systemen is er meestal slechts een beperkte hoeveelheid geheugen (zowel ROM als RAM) aanwezig. Er is in het verleden dan ook al veel onderzoek gedaan naar het automatisch kleiner maken van programma's, zodat ze in het geheugen van een ingebed systeem passen. Dit onderzoek heeft zich tot nu toe vooral toegespitst op het verkleinen van gebruikersprogramma's die op deze systemen draaien, en niet zozeer op het besturingssysteem.

Ingebedde systemen worden steeds complexer, en de ontwikkelaars ervan grijpen dan ook meer en meer terug naar voorgebouwde, generieke componenten die samengevoegd worden om snel de benodigde functionaliteit voor het systeem bij elkaar te brengen. Eén van die componenten is het besturingssysteem. Waar er vroeger vaak een applicatie-specifiek besturingssysteem ontwikkeld werd voor een ingebed systeem, of een bestaand besturingssysteem werd aangekocht dat speciaal ontwikkeld werd voor modulariteit en laag geheugenverbruik, is er tegenwoordig een stijgende interesse voor het gebruik van algemene besturingssystemen zoals Linux in een ingebed systeem. Deze algemene besturingssystemen bieden een aantal voordelen: ze bieden zeer veel functionaliteit aan, ze maken gebruik van standaard interfaces, en er zijn al heel veel programma's voor beschikbaar. Dit alles zorgt ervoor dat de ontwikkelaar zijn apparaat sneller op de markt kan brengen, wat natuurlijk een concurrentievoordeel met zich meebrengt. Het grote nadeel van een algemeen besturingssysteem is dat het veel meer geheugen inneemt: niet alleen biedt het veel meer (vaak onnodige) functionaliteit aan, het is ook niet ontworpen om klein te zijn, maar eerder om algemeen toepasbaar en makkelijk onderhoudbaar te zijn.

In deze doctoraatsverhandeling stellen we een geautomatiseerde methode voor die het geheugengebruik van een besturingssysteemkernel kan beperken met behulp van binaire herschrijftechnieken tijdens

het linken. De technieken zullen voornamelijk nuttig zijn voor het specialiseren van een algemeen besturingssysteem voor een ingebed systeem, waarbij overbodige functionaliteit verwijderd wordt. Linux wordt gebruikt als voorbeeld, maar de aanpak is overdraagbaar naar andere besturingssystemen zoals Windows XP Embedded.

Eerst overlopen we de karakteristieken die besturingssysteemcode onderscheiden van de code van gebruikersprogramma's. Deze karakteristieken hebben meestal te maken met de aanwezigheid van grote hoeveelheden handgeschreven machinetaalcode in de kern. Deze machinetaalcode houdt zich niet noodzakelijk aan dezelfde conventies als code die door een vertaler werd gegenereerd, terwijl een herschrijvende linker juist op die conventies steunt om een een conservatieve maar toch voldoende precieze voorstelling te maken van de code, die geschikt is voor analyse en optimalisatie. We stellen dan ook manieren voor waarop de binaire herschrijver aangepast kan worden zodat ook besturingssysteemcode veilig en efficiënt herschreven kan worden. Eens deze aanpassingen gebeurd zijn, kunnen de gekende compacterende optimalisaties tijdens het linken toegepast worden om al een eerste vermindering van het geheugengebruik te realiseren.

Vervolgens stellen we een aantal specialisaties voor die de besturingssysteemkern aanpassen voor één specifieke hardware/software configuratie. Daarbij wordt allerlei voor deze configuratie overbodige functionaliteit uit de kern verwijderd.

De eerste specialisatie houdt zich bezig met de initialisatiecode en -data in de kern. Linux biedt de mogelijkheid om initialisatiecode en -data uit het geheugen te verwijderen eens het systeem opgestart is en ze niet meer nodig zijn. Het geheugen dat mag vrijgegeven worden wordt door de ontwikkelaars gemarkeerd door middel van manueel aangebrachte annotaties in de broncode. Sommige code en data mag echter slechts in sommige configuraties van de kern vrijgegeven worden na het opstarten, terwijl ze in andere configuraties nog nodig zijn tijdens de verdere uitvoering van het systeem. Deze code en data worden dan ook nooit vrijgegeven. Wij hebben een analyse ontwikkeld die extra initialisatiecode en -data kan opsporen die in een specifieke kernconfiguratie toch mag worden vrijgegeven, hoewel ze door de ontwikkelaars niet als dusdanig was geannoteerd.

Vervolgens wordt het systeemaanroepgedrag van alle gebruikersprogramma's die op het systeem zullen uitgevoerd worden geanalyseerd. Het resultaat van deze analyse is een lijst van alle systeemaan-

roepen die op het systeem kunnen gebeuren, en van alle mogelijke parameterwaarden van deze aanroepen, voor zover het mogelijk was om deze te detecteren. De kern wordt dan gespecialiseerd voor de verworven informatie door de ondersteuning voor alle ongebruikte systeemaanroepen uit de kern te verwijderen, en door de code die de resterende systeemaanroepen afhandelt te specialiseren voor de gekende parameterwaarden.

Veel besturingssystemen bieden de mogelijkheid om bepaalde systeemparameters bij te stellen op het moment dat het systeem gestart wordt, door middel van de *kernel command line*. Op een ingebed systeem heeft de gebruiker normaal gezien geen enkele controle over het opstartproces, en dus ook geen mogelijkheid om deze kernel command line nog aan te passen. Deze vorm van configureerbaarheid is dus overbodig. Onze volgende specialisatie identificeert dan ook de code voor het verwerken van deze command line, en verwijdert ze uit de kern. De kerncode wordt verder ook gespecialiseerd voor de nu als constant te beschouwen waarden van de globale variabelen die door de kernel command line ingesteld konden worden.

Voor onze laatste specialisatie steunen we op de observatie dat het opstartproces van de kern deterministisch is. Tijdens het initialiseren van het systeem is er slechts één draad actief, en die voert steeds exact dezelfde code uit, als we ervan uitgaan dat de hardwareconfiguratie van het systeem nooit verandert. We buiten deze kennis uit door het opstartproces van de kern te observeren, en bij te houden welke intialisatiecode nooit wordt uitgevoerd. Deze code kan dan zonder problemen uit de kern verwijderd worden. Hoewel deze specialisatie weinig of geen invloed zal hebben op het RAM-gebruik van de kern nadat de intialisatiecode en -data uit het geheugen verwijderd zijn, biedt ze de kans om het ROM-gebruik van de kern gevoelig te verkleinen.

Zelfs nadat alle voornoemde specialisaties en compactietransformaties uitgevoerd zijn, blijkt dat een substantieel gedeelte van de kerncode nooit wordt uitgevoerd tijdens de normale uitvoering van het systeem. Een deel van deze onuitgevoerde code is effectief nutteloos, maar kon niet alsdusdanig geïdentificeerd worden door de beperkingen van statische analyse. Dit is niet het geval voor alle onuitgevoerde code: een deel ervan dient voor het opvangen van uitzonderlijke situaties, zoals hardwarefouten, die niet voorkomen tijdens het normale gebruik van het systeem. We kunnen het onderscheid tussen beide categorieën van onuitgevoerde code niet eenvoudig maken, dus kunnen we de onuit-

gevoerde code niet zomaar uit de kern verwijderen. We kunnen het RAM-gebruik van de kern wel verminderen door de onuitgevoerde code slechts in te laden op het moment dat ze effectief nodig blijkt.

We stellen twee benaderingen voor voor het op aanvraag inladen van code: *frozen code compression* en *cold code swapping*. De eerste techniek stelt geen speciale hardwarevereisten, maar is beperkt tot het op aanvraag inladen van onuitgevoerde code. De tweede techniek, waarvoor ondersteuning voor virtueel geheugen vereist is, evenals de aanwezigheid van een snel secundair geheugen (bijvoorbeeld Flash-geheugen), maakt het mogelijk om niet alleen onuitgevoerde code, maar alle zelden uitgevoerde code op aanvraag te laden.

Frozen code compression deelt de onuitgevoerde code op in partities met één ingangspunt, en slaat deze partities op in het geheugen in een gecomprimeerde vorm. In de code worden de partities vervangen door *stubs*. Als het controleverloop zo een stub betreedt, wordt de decompressor opgeroepen die de nodige geheugenruimte alloceert en het corresponderende codefragment decomprimeert. De stub wordt dan overschreven met een spronginstructie die naar de gedecomprimeerde code springt. Eens code gedecomprimeerd is, wordt ze nooit meer uit het geheugen verwijderd.

Cold code swapping gaat anders te werk: zelden uitgevoerde (koude) code wordt gescheiden van vaak uitgevoerde (hete) code en op aparte virtuele-geheugenpagina's geplaatst. De pagina's met koude code worden niet in het fysiek geheugen geladen, maar op een snel secundair geheugen geplaatst. Als een koud codefragment uitgevoerd moet worden, treedt er een paginafout op. De (aangepaste) paginafoutafhandelingscode zoekt dan de corresponderende pagina op in het secundaire geheugen, laadt ze in een vooraf gealloceerde buffer met vaste grootte, en past de paginatabellen aan zodat ze op het virtuele adres in het geheugen gemapt wordt.

De gecombineerde compactietechnieken en specialisaties en het op aanvraag inladen van code slagen er samen in om het RAM-gebruik van de kern terug te dringen met meer dan 48%.

Abstract

There is a lot of research interest in the creation of small programs for embedded systems, where memory (both ROM and RAM) is typically very limited. Until now, this research has focused mostly on reducing the footprint of user space programs, and not on reducing the footprint of the operating system (OS).

Embedded systems are becoming increasingly complex, and developers rely more and more on pre-built, generic components to quickly build the required software functionality for the embedded system. One of these components is the OS. While traditionally embedded system designers created their own OS, specifically tailored to their needs, or used an embedded OS that was specifically engineered for configurability and small memory footprint, there is a growing trend to use general-purpose operating systems like Linux and Windows XP Embedded. However, the use of these general-purpose operating systems entails a lot of overhead, as they were not engineered for small memory footprint, but rather for general applicability and maintainability. The research on compaction and compression of user space programs cannot be applied directly to operating systems, as there are a number of peculiarities in operating systems code that have not yet been taken into account by this research.

In this dissertation, we present an automated way to reduce the memory footprint (both RAM and ROM) of an OS kernel through the use of link-time binary rewriting techniques. The proposed techniques will be useful in particular for streamlining a general-purpose OS kernel for use on an embedded system, removing as much of the aforementioned overhead as possible. We use Linux as a case study, but the approach is transferable to other operating systems as well.

First, we give an overview of the challenges OS kernel code poses to a link-time binary rewriter. Most of these relate to the presence of large

amounts of hand-written assembler code in the kernel that does not necessarily respect the coding conventions a compiler abides by. It are precisely these conventions that a link-time binary rewriter exploits to build a conservative, yet sufficiently precise control flow graph that can be used for analysis and optimization. When the challenges are identified, we present ways for a link-time rewriter to overcome them. Once these challenges are overcome, the well-known link-time compaction optimizations can be applied to the kernel to reduce its memory footprint.

Next, we introduce a number of specialization transformations that allow to adapt the kernel to one specific hardware/software platform, removing functionality that is not needed for this platform.

A first specialization concerns initialization code and data in the kernel. The Linux kernel offers the possibility to remove initialization code and data from memory once the system bootup phase is completed. The code and data to be removed are identified based on manual annotations by the kernel developers. However, some code and data can only be removed after bootup in some kernel configurations, but not in others. Consequently, this code and data is never removed from memory. We have developed an analysis that can identify extra initialization code and data that was not annotated by the kernel developers.

Next, the system call behavior of all user space programs that will run on the system is analyzed. This analysis results in list of all system calls that can be used on the system, and, in case they could be identified, all possible values for the arguments of these system calls. The kernel is then specialized for this knowledge by removing all unused system call handlers, and by specializing the code of the remaining handlers for the known parameter values.

Many OS kernels provide the ability to tweak certain system parameters at boot time through a so-called kernel command line. On embedded systems, where the user typically has no influence over the boot process, this configurability is useless. Therefore, our next specialization consists of identifying the command-line handling code and removing it from the kernel, and specializing the kernel code for the now known-to-be-constant values of the global configuration variables that are initialized through the kernel command line mechanism.

As a last specialization, we observe that the kernel's boot process is deterministic. During system initialization, there is only one thread

active, and, supposing the system's hardware configuration does not change over time, the same code is executed at every system boot. We exploit this knowledge by observing the kernel's boot process and recording which code is executed. All unexecuted initialization code is then removed from the kernel. While this specialization has only minimal impact on the kernel's RAM footprint, it has the potential to significantly reduce the ROM footprint.

Even after all aforementioned compaction and specialization transformations have been applied, code coverage analysis shows that a substantial portion of the kernel code is never executed during normal system operation. While part of the unexecuted code is in effect useless, but could not be proven so because of the limitations of static analysis, there is no way to distinguish this code from the also unexecuted, but necessary, code for handling exceptional situations such as hardware failures. Even though the unexecuted code cannot be removed from the kernel entirely, the kernel's RAM footprint can be reduced significantly by introducing an on-demand code loading scheme that only loads this code when it is needed.

We propose two approaches to on-demand code loading: frozen code compression and cold code swapping. The first technique has no special hardware requirements, but is limited to loading unexecuted code on demand. The second technique, which requires virtual memory support and the presence of a fast secondary memory (e.g., Flash memory) in the system, makes it feasible to load all infrequently executed code on demand, not just the code that is never executed under normal operation.

The frozen code compression technique divides the unexecuted kernel code into single-entry partitions that are stored in compressed form, and replaced by stubs in the code. When control flow enters a stub, a decompressor is invoked that allocates space and decompresses the corresponding code fragment, after which the stub is overwritten with a direct jump to the decompressed code. Once decompressed, code is never evicted from memory.

The cold code swapping technique places the infrequently executed (cold) code together on virtual memory pages that are separate from those containing frequently executed (hot) code. The cold code pages are not loaded in physical memory, but stored on a fast secondary storage medium (e.g., Flash memory). Whenever a cold code fragment needs to be executed, a page fault will occur. The kernel's (modified)

page fault handler will then locate the corresponding page on the secondary storage medium and load it in a fixed-size buffer, mapping it at the correct address in virtual memory. This scheme allows for eviction of loaded code when the buffer is full.

Combined, the proposed compaction, specialization and code loading techniques reduce the kernel's RAM footprint with up to 48%.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In most embedded systems, resources like processor cycles and available memory (both RAM and ROM) are constrained. There are several reasons for this, most notably power consumption and unit cost. Power consumption is very important for battery-operated devices, and lower-clocked processors and smaller RAM chips can mean an important improvement for the device's battery lifetime. Unit cost comes into play for mass-produced devices. Even small cost savings on the components of a system, e.g., one fewer RAM chip or a smaller Flash ROM, add up to significant savings when enough devices are manufactured, which is important in the very cost sensitive embedded market. It is therefore not surprising that embedded system designers pay much attention to program size.

Over time, the complexity of embedded systems has risen dramatically. A prime example of this trend are mobile phones. Over the years, these devices have evolved from providing simple telephone calls to gadgets that also send text messages, take pictures, play music and surf the internet. The traditional methods for creating small programs, i.e., writing them in assembler or manually fine-tuning the compiler output are no longer practical for such complex systems, especially in highly competitive markets, where time-to-market is very important. Nowadays, embedded systems designers are turning to pre-built components and higher-level languages to manage the complexity and to deliver products on time. Of course, this approach results in some overhead: pre-built components are usually more generic than they have to be for one specific application, and high-level languages do not always translate easily to compact assembler code. Not surprisingly, a lot of

research has been done on ways to mitigate the size overhead incurred by the use of high-level abstractions [Besz03].

1.1 Reducing Program Footprint

As indicated before, in an embedded system there are two different scarce resources that are impacted by program size: RAM and ROM. Optimizations that reduce a program's use of one resource are not necessarily beneficial in reducing the use of the other resource. For example, transforming the program into a self-extracting executable would limit the ROM size, but would have no influence at all on the RAM usage. Conversely, an optimization that reduces the amount of dynamic memory allocated by the program will surely reduce the RAM usage but will do nothing for the ROM size. Therefore it is useful to define several different program footprints:

- **RAM footprint** is the number of bytes a program occupies in RAM memory. This can be further subdivided into *static* RAM footprint, i.e., the sum of the sizes of the program code and statically allocated data, and *dynamic* RAM footprint, which is the sum of the static RAM footprint and the maximum amount of dynamically allocated memory (stack + heap memory) during program execution.

- **ROM footprint** is the number of bytes a program takes in ROM memory. This is different from the static RAM footprint because this includes the size of the program headers (defined by the executable format) and excludes the size of the zero-initialized data, which is typically not stored with the program image. Furthermore, it is quite common to store programs in a compressed format, which significantly reduces the ROM footprint but not the static RAM footprint.

Static RAM footprint and ROM footprint are well suited for reduction through automated methods. By and large, the footprint reduction techniques can be divided into two types: *compaction* and *compression* [Besz03]. The end result of compaction is a smaller program that is still directly executable. With compression this is not the case: an additional decompression step, either in hardware or in software, is necessary before the program can be executed. Note that it is not

necessary to decompress the whole program at once: many techniques selectively decompress code or data whenever they are needed. Typically, compression techniques will be more efficient size-wise, but with a larger execution time penalty because of the inevitable decompression. Compaction and compression are orthogonal: because the end result of compaction is still an executable program, one can still apply compression techniques to a compacted program, combining the advantages of both approaches.

Several techniques exist to reduce the heap memory usage, and thus the dynamic RAM footprint, of a program through compression of the dynamically allocated data [Chen03a, Latt05, Zhan06b]. These techniques can be applied on top of the aforementioned compaction and compression transformations that reduce the static RAM and ROM footprint. The heap compression techniques are, however, beyond the scope of this dissertation, which will focus solely on techniques for reducing the static RAM and ROM footprint.

1.2 Broadening the Scope

A common characteristic of the existing compaction and compression techniques (Beszédes et al. give a comprehensive overview [Besz03]) is that they optimize individual programs independently. However, embedded systems software usually consists of more than just one program. Figure 1.1 gives an abstracted overview of the software components of a complete system. The main software components are applications, libraries and the operating system (OS). The libraries offer common functionalities that can be used by the different applications. Underlying all this is the operating system which acts as a gateway to the system's hardware. The operating system offers a range of services to the applications, ranging from access to the hardware devices over interprocess communications (IPC) to memory protection for the running processes. There is a lot of interaction between the different components: applications call code from libraries and both applications and libraries can request services from the operating system. Furthermore, there are several ways in which applications can communicate, either via IPC or directly through shared memory.

Not every embedded system has all these components. In particular, simpler embedded systems like washing machines or microwave ovens are unlikely to have an operating system or even separate ap-

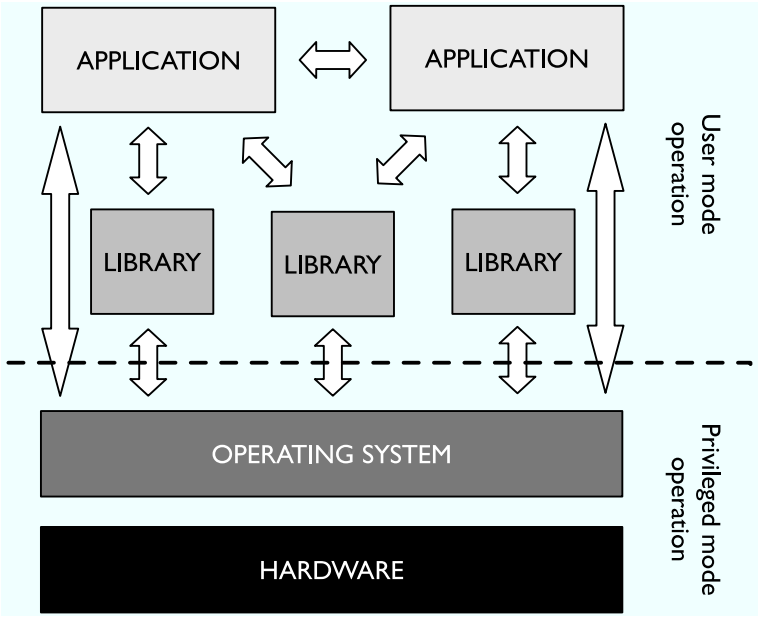


Figure 1.1: An abstracted overview of a complete system.

plications and libraries. The functions performed by these machines are simply not complex enough to warrant the overhead of an extra abstraction layer in software. For these systems, the software component can be viewed as a single application to which compaction and compression techniques can be applied.

However, a very large class of embedded systems does have all these different software components. A survey of embedded system developers shows that in 2006 81.3% of all embedded system designs incorporated some form of operating system [Turl06]. For these systems, it is of course possible to apply compaction and compression techniques to each component in isolation, but better results can be achieved if the system is analyzed and optimized as a whole. There are several reasons for this:

- In a complex system, not all components are written from scratch. In particular both the OS and some libraries, maybe even some applications, may have been bought from third-party vendors and will just be integrated into the system. For reusability, these pre-built components are more general than needed for any one

specific system in which they are used. By looking at all components of the system at once, it is possible to specialize them for their specific uses in this system.

- Because the components are written separately, they need to abide to certain conventions to make sure they work together properly. These conventions are codified into the system's application binary interface (ABI). The ABI specifies interfaces between applications and libraries such as which processor registers are considered callee-saved and which are caller-saved, as well as the interface between applications and the operating system. This ABI is usually uniquely defined for each combination of operating system and processor family. These conventions are optimized for the general case, even though in some cases programs would be faster or more compact if they could ignore the ABI. Once all components in the system are known, it is possible to perform a *whole-system integration* step that removes the restrictions of the ABI and integrates the components in a more efficient way.

The idea of whole-system optimization is especially attractive on the very large class of embedded devices for which the functionality is known and fixed over the lifetime of the device. Examples of such systems include wireless internet routers like the Linksys WRT54G or hard-disk video recording systems like the TiVo. Even if the software on these systems (usually called *firmware* in this context) has to be updated during the system's lifetime, this happens in an atomic operation: all the firmware is updated at once. Consequently, there are never problems with adding some new functionality to the device that would require some feature from a library that was stripped out because it was not needed before. After the feature is added to the firmware, the whole-system integration step is rerun, and in the new firmware image the necessary library functionality is included as well.

The previous discussion outlines the ideal situation in which the whole system can be optimized at once. However, there is still a long way to go before this ideal can be realized. Only recently the first paper on the subject of a unified whole-system representation for analysis and optimization has appeared [Bert06]. Before we can co-optimize the whole system, it is necessary that all individual components can be modeled and optimized appropriately. While the current state of the art is already well suited for the modeling and optimization of user space

applications, this is not the case for the operating system. Some of the assumptions that can be made in the modeling of regular applications do not hold for the OS because it runs in the *privileged* protection domain, instead of the *user* protection domain, as shown in Figure 1.1. There will also be some unique opportunities for compacting the OS code that are not available for regular applications.

In this dissertation, we will investigate the difficulties in modeling the operating system and its interactions with the user space programs and propose techniques to reduce the static RAM and ROM footprint of the operating system as a step on the road to the ideal of whole-system optimization. It is important to note that the term "operating system" in this context refers to the OS *kernel*, the code that runs in the privileged protection domain. In general usage, the term operating system is used to denote the combination of this kernel and a number of companion applications like the command-line shell or the graphical user interface. These companion applications are in fact just regular programs that behave like any other user space program and can thus already be modeled and optimized using existing techniques.

1.3 Our Approach to OS Kernel Footprint Reduction

There are three levels at which optimization of the OS kernel can be performed: at the source code level, at the compiler level, or at the linker level. Each level has its advantages and disadvantages.

At the source code and compiler levels, the greatest amount of semantic information is available, as this information can be derived directly from the source code. Furthermore, source code level optimizations and optimizations in the compiler front end are architecture independent. However, there are significant disadvantages associated with optimization at this level. First, there is no whole-program overview, which limits the scope of the optimizations, and makes it difficult to optimize low-level aspects of the kernel, for example, by modifying the calling conventions for specific procedures. Secondly, the techniques are source language dependent. Porting the developed techniques to an OS kernel that is written in a different programming language involves a lot of effort, as all optimizations have to be ported to a new source-level transformation tool or a new compiler. Finally, OS kernels

contain a significant amount of hand-written assembler code for tasks that simply cannot be expressed in higher-level languages, e.g., accessing the processor's control registers. This assembler code will not be analyzed or optimized by either a source-level transformation tool or a compiler.

At the linker level, the main advantages are the whole-program overview, which also includes all hand-written assembler code, and the complete source language independence of the techniques. The drawbacks of link-time optimization are the absence of high-level semantic information (e.g., variable types and procedure signatures), and the fact that the techniques are architecture-dependent as the optimizations operate directly at the machine code level. Recent research, however, offers workarounds for the dearth of semantic information at the linker level, either by using a hybrid approach where certain analyses are performed at the source code level [He07], or by using the debug information produced by the compiler as extra input to the link-time optimizer [VanP07a]. Furthermore, as was shown by De Bus [DeBu05], and is illustrated in Diablo, the link-time optimizer we have developed, it is possible to build a link-time optimizer in such a way that it is retargetable to new processor architectures.

After taking the aforementioned advantages and drawbacks of each optimization level into consideration, we have decided to situate our work at the linker level.

1.3.1 Link-time Compaction

An OS kernel contains a lot of performance-critical code. For instance, the interrupt handlers should complete their work as soon as possible. If handling an interrupt takes too long, data may get lost for devices like high-speed network interfaces that have a high interrupt firing rate. Therefore it is important that the performance impact of any compaction or compression transformation applied to the kernel is very limited, or at least that these performance-critical parts of the kernel are not adversely affected by the transformations.

With this in mind, we have chosen to concentrate first on link-time compaction techniques as a way to reduce the kernel's memory footprint. Most link-time compaction techniques (De Sutter et al. give an overview of the most useful link-time analyses and optimizations [DeSu05]) do not have a negative impact on execution speed, as

they mostly remove unreachable code or unnecessary computations. In fact, by removing these unnecessary computations, it is even possible to achieve some speedups. The big exception to this rule are the duplicate code elimination techniques, in particular those that extract code fragments at a granularity smaller than whole procedures as these introduce extra procedure calls and returns, with the associated overhead. However, this speed impact can be mitigated by using profile information to identify the frequently executed code, which is then excluded from this transformation [DeSu02].

In order to reliably perform link-time binary rewriting, it is imperative that the rewriting tool is able to construct a correct, conservative control flow graph of the program [Debr00, DeBu05, DeSu05]. As mentioned before, an OS kernel is somewhat different from a regular user space program. As such there are some issues that have to be taken into account when building the kernel's control flow graph, most of which have to do with the aforementioned presence of hand-written assembler code in the kernel. We will identify these issues, and present ways to tackle them.

1.3.2 Operating System Kernel Specialization

By exploiting knowledge about an embedded device's hardware and software configuration, we can achieve size reductions on top of those already achieved with the established link-time optimizations. For almost all embedded systems, the hardware is fixed over the lifetime of the device. For a smaller, but still very large fraction of systems this is also the case for the software that will run on the device. Consequently, it makes sense to specialize the operating system for the specific hardware/software combination of a device. The first avenue to explore for this specialization is of course the operating system's build time configuration system. This allows developers to cherry-pick, for example, which hardware drivers will be compiled into the kernel, which filesystems should be supported, whether a TCP/IP stack should be included, et cetera. Operating systems that are specifically designed for use in embedded systems, like eCos[1] or vxWorks[2] have a very fine-grained configuration system, precisely because they have been designed from the ground up for use in environments with limited mem-

[1] `http://ecos.sourceware.org/`
[2] `http://www.windriver.com/vxworks/`

ory. This is less so for general-purpose OS kernels like Linux that have been adapted for use in embedded systems only later in their development history. The main focus in the development of these kernels is on generality and maintainability of the code base, with modularity and compactness only as secondary design goals. For example, the Linux configuration process does not allow the developer to choose which system calls should be included in the kernel, even though on a fixed-functionality system it is possible to determine at design time exactly which system calls will be used by the software.

On top of the established link-time compaction techniques, we have developed a number of automated specialization transformations that will adapt a kernel to a specific hardware/software combination. This approach will have the most impact on a general-purpose OS like Linux, where there is more room for improvement, but some of the techniques we introduce will work equally well on operating systems that do have a very fine-grained configuration system.

The proposed specializations, which are all applicable to the Linux 2.4 kernel, will remove unnecessary system calls from the kernel, and specialize the remaining ones for known constant arguments. The boot-time configurability that is present in some operating systems, but is typically useless for embedded systems, will be removed, and the kernel code will be specialized for the now constant values of the parameters that could be tuned through the boot-time configuration interface. A further specialization identifies initialization code and data that are no longer needed in the kernel once the system has finished booting, and allows these to be removed from memory after bootup. The last specialization also concerns the aforementioned initialization code. As long as a system's hardware configuration is unchanged, the boot process is deterministic. Through instrumentation, we observe the kernel's boot process and record which part of the initialization code remains unexecuted. This code, and its associated data, may then be removed entirely from the kernel. While this does not reduce the kernel's RAM footprint (the initialization code would have been removed from memory after bootup anyway), this specialization is useful for reducing the kernel's ROM footprint.

1.3.3 On-demand Code Loading Techniques

Even after all aforementioned compaction and specialization transformations have been applied, code coverage analysis shows that a substantial portion of the kernel code is never executed during normal system operation. Part of this unexecuted code is truly unreachable, but could not be detected by the compaction and specialization techniques due to the limitations of static analysis. The rest of the unexecuted code is necessary for the handling of exceptional situations, for example hardware failures. As we cannot make the distinction between both kinds of code through static analysis, the unexecuted code cannot be removed from the kernel. It is, however, possible to introduce an on-demand code loading scheme that loads only the necessary code into memory. This way, we can reduce the kernel's static RAM footprint without compromising its reliability by removing potentially necessary code.

We will introduce two approaches to on-demand code loading that take into account the specific nature of an OS kernel. The first technique, *frozen code compression*, uses compression to reduce the footprint of unexecuted code. Code fragments are decompressed on an as-needed basis. To avoid concurrency issues, decompressed code fragments are never evicted from memory. The advantages of this technique are that the performance impact is limited, and that it requires no special hardware support. The biggest disadvantage is that, because of the no-eviction policy, it is impossible to predict the exact amount of memory that will be used by the kernel code.

The second technique, *cold code swapping*, requires support for virtual memory. Infrequently executed (cold) code is placed on separate virtual memory pages from the frequently executed code. The cold code pages are not present in physical memory. Instead, they are stored on a fast secondary storage medium (e.g., Flash memory). Whenever a cold code fragment needs to be executed, a page fault will occur. The kernel's (modified) page fault handler will then retrieve the needed page from the secondary storage, and map it into a fixed-size buffer in physical memory. We will show that this approach allows us to evict pages from memory without having to add additional locking in the kernel to avoid concurrency issues. In order to minimize the number of code loading events, and thus the performance impact, intelligent code placement algorithms are needed for the cold code. We will discuss and evaluate two such algorithms. This technique has two main ad-

vantages: it allows for on-demand loading of all infrequently executed code, not just the never-executed code, and it is possible to predict exactly how much memory will be needed for the kernel's code. The main disadvantages are the potentially larger performance impact, and the fact that support for virtual memory is required.

1.4 Linux as a Case Study

The techniques described in this work are generally applicable to all operating system kernels, and in particular to general-purpose operating systems used in the context of embedded systems. For both the discussion and evaluation of the techniques we will focus on the Linux 2.4 kernel.

On the one hand, this choice is motivated by practical reasons. Both the source code for the kernel and a whole body of information on its inner workings are publicly available, which simplified the study of the system. Furthermore, the link-time binary rewriting framework we used to implement our techniques requires the presence of not only the executable kernel image but also the object files from which this image is constructed. Consequently, recompiling the kernel was necessary, which is of course only possible when the source code is available.

On the other hand, the application of the proposed techniques to the Linux kernel is also quite relevant. In 2005, 24% of all embedded systems designers used Linux as their OS according to a survey taken by the Embedded Systems Design magazine [Turl05]. In the same survey, memory usage is shown to be the second most important technical objection against Linux, the most important being the lack of hard real time capabilities or low performance of Linux in general. Consequently, we can conclude that there is a big interest in the use of Linux on embedded systems, and reducing Linux' memory footprint can help increase its relevance in this market.

1.5 Major Contributions

The major contributions presented in this dissertation are:

- We identify a number of OS kernel code peculiarities not encountered in user-space programs, and present ways to model them

in a link-time binary rewriter, allowing for conservative, yet sufficiently precise, analysis and optimization.

- We present an automated method for specializing the OS kernel for a specific hardware/software combination, thus improving on the OS kernel's build-time configuration system.

- We explore how the kernel's code memory footprint can be reduced in an automated way by introducing on-demand code loading schemes that target code for the handling of exceptional situations and other infrequently executed code. The presented techniques take the specific nature of an operating system kernel into account, and deal reliably with concurrency while still maintaining good performance.

- We have evaluated the proposed techniques on two different platforms, i386 and ARM, and thus demonstrated the feasibility of our approach.

- The proposed techniques were implemented in a proof-of-concept OS kernel compaction tool based on the Diablo link-time binary rewriting framework, which is used internationally for research on link-time optimization [Gilb06, Bans06], software security [Hu06], program debugging [Gupt05, Zhan06a], binary obfuscation [Mado06], program watermarking [Anck04], etc. In the course of this implementation work, a number of core changes to Diablo were made that resulted in an overall better reliability and extensibility of the framework. As such, this work has indirectly contributed to advancements in the aforementioned research fields.

We feel this work is the first systematic exploration of techniques to reduce the memory footprint of an operating system kernel at link time, and a milestone on the way to the ideal of whole-system optimization as presented in Section 1.2.

1.6 Publications

Three publications are directly connected to the OS kernel footprint reduction work described in this dissertation. The work on OS kernel specialization was first presented at the 2005 Conference on Languages,

Compilers and Tools for Embedded Systems (LCTES) [Chan05]. An extended version of this work, which also presented the first of our on-demand code loading schemes, frozen code compression, was published in the ACM Transactions on Embedded Computing Systems journal in 2007 [Chan07b]. The second code loading technique, cold code swapping, is described in a paper that has been accepted for publication in Transactions on HiPEAC [Chan07a].

In a broader context, but still related to the work presented in this dissertation, the author has contributed to several papers on the subject of link-time binary rewriting and link-time compaction, published in international journals and presented on international conferences. [DeBu03, DeBu04b, VanP05b, VanP05a, VanP07b, DeSu07]

Apart from the work described in this dissertation, the author has also contributed to research on the application of link-time binary rewriting in the fields of program instrumentation [DeBu04a] and software security [Anck04].

Below is a full list of all publications on international conferences and in international journals the author has contributed to:

- De Bus, B.; Kästner, D., Chanet, D.; Van Put, L.; De Sutter, B. *Post-Pass Compaction Techniques*. Communications of the ACM. [DeBu03]

- De Bus, B.; Chanet, D.; De Sutter, B.; Van Put, L.; De Bosschere, K. *The Design and Implementation of FIT: a Flexible Instrumentation Toolkit*. Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04). [DeBu04a]

- De Bus, B.; De Sutter, B.; Van Put, L.; Chanet, D.; De Bosschere, K. *Link-Time Optimization of ARM Binaries*. Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04). [DeBu04b]

- Anckaert, B.; De Sutter, B.; Chanet, D.; De Bosschere, K. *Steganography for Executables and Code Transformation Signatures*. Information Security and Cryptology - ICISC 2004. [Anck04]

- Chanet, D.; De Sutter, B.; De Bus, B.; Van Put, L.; De Bosschere, K. *System-Wide Compaction and Specialization of the Linux Kernel*. Proceedings of the 2005 ACM SIGPLAN/SIGBED Confer-

ence on Languages, Compilers and Tools for Embedded Systems (LCTES'05). [Chan05]

- Van Put, L.; Chanet, D.; De Bus, B.; De Sutter, B.; De Bosschere, K. *DIABLO: a reliable, retargetable and extensible link-time rewriting framework*. Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology. [VanP05a]

- Van Put, L.; De Sutter, B.; Madou, M.; De Bus, B.; Chanet, D.; Smits, K.; De Bosschere, K. *LANCET: a nifty code editing tool*. Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05). [VanP05b]

- Van Put, L.; Chanet, D.; De Bosschere, K. *Whole-Program Linear-Constant Analysis with Applications to Link-Time Optimization*. Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems. [VanP07b]

- De Sutter, B.; Van Put, L.; Chanet, D.; De Bus, B.; De Bosschere, K. *Link-Time Compaction and Optimization of ARM Executables*. ACM Transactions on Embedded Computing Systems. [DeSu07]

- Chanet, D.; Cabezas, J.; Morancho, E.; Navarro, N. *Linux Kernel Compaction through Cold Code Swapping*. Transactions on HiPEAC. [Chan07a]

- Chanet, D.; De Sutter, B.; De Bus, B.; Van Put, L.; De Bosschere, K. *Automated Reduction of the Memory Footprint of the Linux Kernel*. ACM Transactions on Embedded Computing Systems. [Chan07b]

1.7 Outline

This dissertation is organized as follows. Chapter 2 gives a brief overview of link-time binary rewriting, introducing the terms and concepts that will be used throughout the rest of the thesis. The next chapter details the challenges we faced in adapting a link-time binary rewriter for user space programs for use on an operating system kernel. Chapter 4 introduces techniques for specializing an OS kernel for a specific hardware/software combination. In Chapter 5, on-demand

code loading techniques are explored in the kernel context. We discuss related work in Chapter 6 and draw conclusions and discuss future work in Chapter 7.

Chapter 2

An Introduction to Link-time Binary Rewriting

In this chapter, we will first briefly explain how a traditional linker works. Next, we show how a link-time binary rewriter uses the available information to build a whole-program representation that is suitable for conservative analysis and optimization. Then, we will address in detail the reliability issues involved in link-time program rewriting. We conclude with an overview of existing link-time analyses and optimizations aimed at program compaction.

2.1 Traditional Linking

The basic task of a linker is to combine the compiled and assembled source code files (called *object files*) and libraries into an executable program. A library is in essence just a collection of object files. There are two different ways in which a program can be linked: statically or dynamically. In the case of static linking, the final program is standalone: it contains all code and data necessary for execution. In dynamically linked programs, the libraries are not linked into the program, but are separate files that are shared between different programs. This makes sense for user space programs, as it avoids unnecessary duplication of library code and data both on disk and in memory. However, as the operating system kernel is always statically linked, we can concentrate on static linking in the remainder of this dissertation.

As shown in Figure 2.1, both object files and executable programs

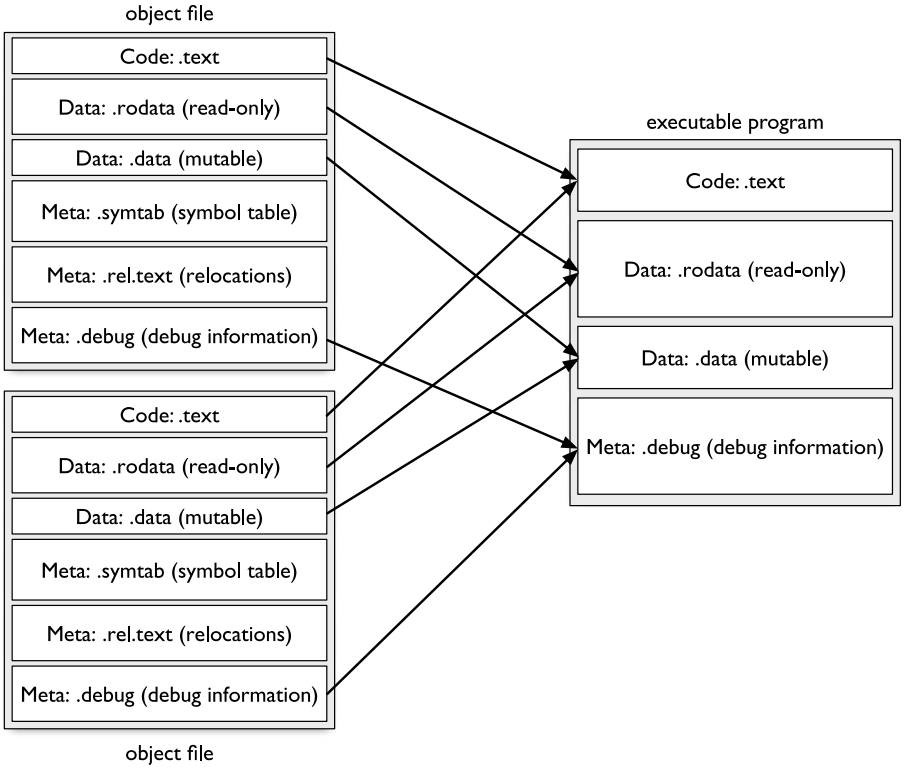


Figure 2.1: Object files are combined into an executable program by the linker.

are structured as a collection of sections. There are different kinds of sections:

- *Code sections*: these contain executable program code. There may be read-only data intermingled with the code as well.
- *Data sections*: these contain the program data. They can be read-only (e.g., constant strings, jump tables) or mutable (e.g., global variables).
- *Meta data sections*: these contain meta data that is used by the tool chain to create the final executable program. Not all of the meta data is needed for the actual execution of the program, so only part of it will appear in the final executable.

The meta data warrants further discussion. It consists mostly of:

- *Symbol information*: this information is used to identify entities in the code and data, for example procedures and global variables. A symbol can be interpreted as a label attached to such an entity. Symbols have either local or global scope: local symbols are only visible within the object file in which they are defined (this is useful for, e.g., static functions or variables), whereas global symbols are externally visible as well.
- *Relocation information*: as the final addresses of the code and data sections are not yet known at compile time, the compiler has to insert placeholder values for each symbol reference in the code and data. The relocation information contains a list of all locations in the object file where such placeholder values are inserted, and details how the final value should be computed (i.e., which symbol is referenced, and what computation should be done on this symbol's final address to fill in the placeholder). Entries in the relocation information will from now on simply be called *relocations*.
- *Debug information*: contrary to the previous two meta-data types, the debug information is not included in the object files for the benefit of the linker, but for use with a symbolic program debugger. This information provides an additional mapping between the generated machine code and the original source code.

The linker generates the final program in three phases:

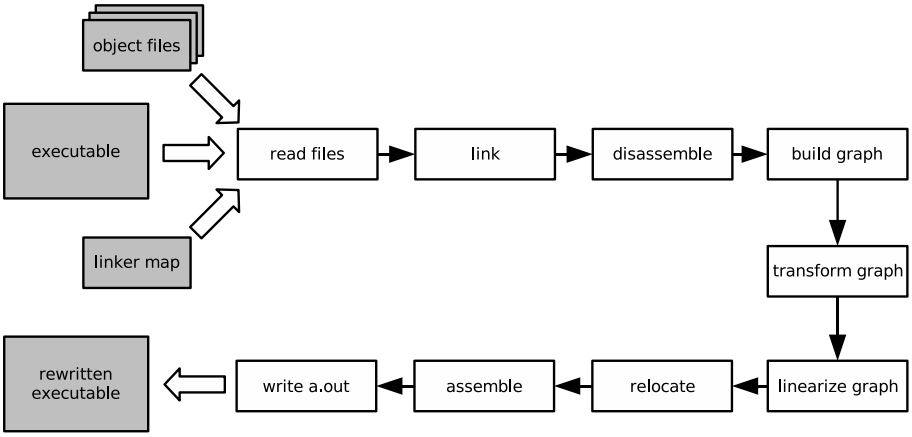


Figure 2.2: Overview of the operation of a link-time binary rewriter.

1. Based on the symbols referenced in the relocation information, and the symbols declared by the different object files, the linker decides which object files from which libraries have to be included in the final program. Object files that are passed directly as inputs to the linker, not as part of a library, are always entirely included in the program. This phase is called *symbol resolution*.

2. By default, identically named sections from different object files are combined into one big section in the final program. This rule can be deviated from in most linkers through the use of a so-called linker script that defines additional rules for combining and placing sections. These sections are then assigned their final addresses.

3. Using the relocation information, the linker can now recompute all placeholder values and replace them with their final value.

While we presented only a simplified version of the static linking process, this is sufficient to demonstrate the role of the symbol and relocation information. A more in-depth description of the linking process can be found in the book “Linkers & Loaders” by Levine [Levi00].

2.2 Link-time Binary Rewriting

A link-time binary rewriter has the same information at its disposal as a regular linker, i.e., the object files and libraries, and the symbol and relocation information that is contained in those files. Figure 2.2 summarizes the operation of a link-time binary rewriter. This figure is based on the inner workings of Diablo[1] [DeBu04b, VanP05a, DeSu07], a link-time binary rewriting framework we developed at the ELIS department of Ghent University. This is the framework we have used to evaluate the techniques presented in this work.

First, the regular linker is used to create an executable program and a map file. This map file details the order in which the object file's sections are laid out in the executable program. Diablo then uses these two files as extra inputs, and in a first step re-links the program in exactly the same way as the original linker. This allows the link-time rewriter to capture all possible information on the executable, including the aforementioned information stored in the object files as well as any information added by the regular linker itself. Next, the linked program's code sections are disassembled, and a whole-program control flow graph is built. This representation, which we will describe in more detail in Section 2.3, is used for all analyses and transformations on the program. After all transformations are applied, the code is again converted into a linear form during the graph linearization phase. Next, the linear code is assembled, all relocations are recomputed, and the rewritten program is stored on disk.

2.3 The Augmented Whole-program Control Flow Graph

Based on the information available in the object files, a suitable intermediate representation of the program to be rewritten has to be constructed. Most link-time rewriters operate on a whole-program control flow graph (WPCFG), which consists of the combined CFGs of all procedures in the program. In link-time WPCFGs, the intermediate instructions are usually very close to the actual machine code instructions, and they operate on registers as if they are global variables, whereas memory is treated as a black box. This very conservative treat-

[1] `http://diablo.elis.ugent.be/`

ment of memory accesses is due to the fact that at this low level, where all source code information about variables and their types is absent, it is hard to perform accurate alias analysis [Hind01].

To model indirect control flow elegantly, a virtual *unknown node* is usually added to the WPCFG. The relocation information from the object files is used to identify all possible targets of indirect control flow: if a procedure is never referenced from data or code, there is no way in which its address can be produced at run time and used as a target of indirect control flow. An important exception to this rule, computed jumps, will be discussed in Section 2.4. All basic blocks that are possible targets of indirect control flow become successors of the unknown node, and all basic blocks ending in indirect control transfers become its predecessors. By imposing conservative properties on the unknown node, it is then possible to handle unknown control flow conservatively in any of the applied program analyses and transformations. For liveness analysis, for example, the unknown node is defined as reading and writing all registers. An example of the use of the unknown node is depicted in Figure 2.3.

Instead of using a simple WPCFG, Diablo uses an Augmented WPCFG or AWPCFG. Besides nodes modeling the program's basic blocks, the AWPCFG also contains nodes for all data sections in the object files, such as the read-only, zero-initialized or mutable data sections, the global offset table section, etc. Furthermore, the edges in the graph are not limited to the control flow edges that model possible execution paths. Additionally, the AWPCFG contains *data reachability edges* that directly represent the relocation information from the object files. For example, an instruction computing a relocatable address of some data section will be connected to the node corresponding to that section. Likewise, if the relocatable address of some data or instruction in node A is stored in data section B, a data reachability edge from B to A will be present. As such, the data reachability edges model code/data that is reachable/accessible indirectly through computed jumps or indirect memory accesses.

The program data is represented in this graph at the granularity of object file sections (henceforth called *subsections*, as opposed to the sections in the executable program). In general, the more fine-grained the data nodes in the AWPCFG are, the more accurate analyses will be and the more aggressive optimizations like unreachable data removal can be. At link time, data section nodes in the AWPCFG can

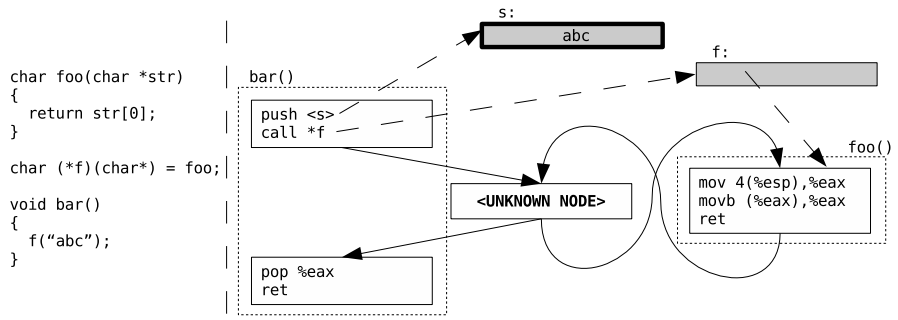


Figure 2.3: An example AWPCFG.

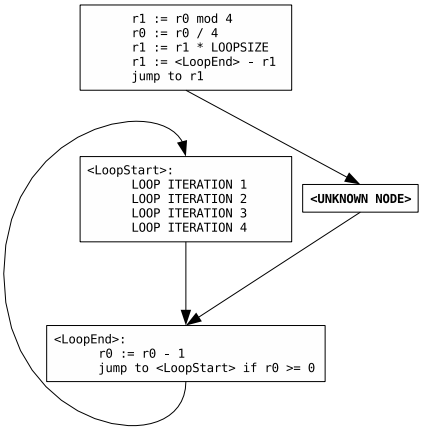
in general not be split into smaller nodes, because the compiler might have performed base-pointer optimizations on different pointers to the same section. There is an important exception to this assumption, however, which concerns constant strings, such as the format strings used for C-language procedures such as `printf`, in GCC-compiled object files. On platforms that use the ELF file format [Nohr93] (e.g., Linux and most modern UNIX platforms), these constant strings are collected in `.rodata.str` sections, and they are hence easily detected. Because the GCC compilers only generate direct accesses to these strings, any `.rodata.str` section of an object file containing multiple strings can be safely split into multiple sections and hence multiple nodes in the AWPCFG.

Figure 2.3 shows an example AWPCFG. On the left, a source code fragment is shown, on the right the corresponding AWPCFG is pictured. Solid edges are control flow edges, dashed edges are data reachability edges. The gray blocks represent the data sections in the program; those with a thick border contain read-only data, the others contain mutable data. These representation conventions will be used throughout the rest of this dissertation.

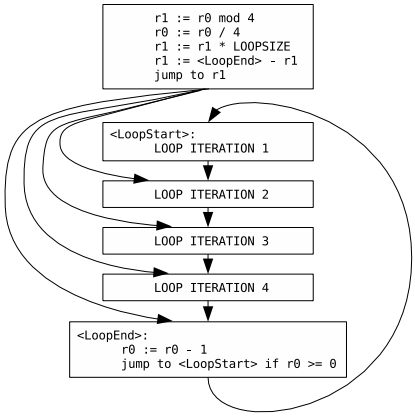
In the source code fragment, `bar` calls `foo` through a function pointer. In essence, `bar` is a caller of the unknown node and `foo` is a callee from the unknown node. In real programs, there are numerous such callers and callees. The first node of `bar` references both `f` and the string `"abc"`, so there are data reference edges from this node to both data nodes. Because the variable `f` holds the address of `foo`, a data reachability edge points from `f` to `foo`. This edge corresponds to the control flow edge from the unknown node to `foo`.

```
      r1 := r0 mod 4
      r0 := r0 / 4
      r1 := r1 * LOOPSIZE
      r1 := <LoopEnd> - r1
      jump to r1
<LoopStart>:
      LOOP ITERATION 1
      LOOP ITERATION 2
      LOOP ITERATION 3
      LOOP ITERATION 4
<LoopEnd>:
      r0 := r0 – 1
      jump to <LoopStart> if r0 >= 0
```

(a) pseudo code



(b) naïve control flow graph



(c) correct control flow graph

Figure 2.4: Computed jumps in link-time binary rewriters.

2.4 Reliability

The reliability of a link-time binary rewriting framework depends entirely on the reliability of its underlying program representation. The AWPCFG has to be conservative: it should represent at least all possible execution paths in the program. A number of recent publications [DeBu05, DeSu05, DeSu07] have described in detail how a conservative AWPCFG can be constructed by using the relocation information available at link time and by using pattern matching. The latter is required for computed jumps, especially in hand-written assembly code or in position independent code (PIC), because the behavior of such jumps is not defined in enough detail by the available relocation information. An example is shown in Figure 2.4. Part (a) shows a fourfold unrolled

loop in pseudo code. At the beginning of the code fragment, `r0` holds the original loop counter. `r0` is divided by four, and `r1` is used to determine where in the unrolled loop the iteration has to start to compensate for loop counts not divisible by four. Using only the relocation information, the link-time rewriter would construct a control flow graph as depicted in part (b) of the figure. The basic block ending in an indirect jump is a predecessor of the unknown node, and the block labelled `LoopEnd` is made a successor of the unknown node due to the relocation appearing in the fourth instruction of the pseudo code. The four unrolled loop iterations are placed in the same basic block, which is obviously incorrect, as the indirect jump in the top block can transfer control to any of the individual unrolled loop iterations. The correct modeling of this code is shown in part (c) of the figure.

To find the potential targets of such control flow transfers, Diablo relies on pattern matching. Whenever a use of the program counter or some unconventional control flow transfer is detected of which it is uncertain how it functions, the surrounding program fragment or program slice is compared to a number of patterns. To make this work reliably, the patterns to which a fragment is compared must be such that they each define a specific behavior unambiguously. For example, a pattern that matches address table lookups (commonly used in the implementation of C-language switch statements) should include the necessary boundary checks that check for constant values. Only if these boundary checks can be found, and consequently the boundaries of the lookup table can be computed, it is possible to determine all potential targets of the computed jump.

More generally, the term "unambiguously" here means that a pattern should be such that the behavior of a matched fragment is known well enough to build a conservative program representation that is still precise enough to be useful. In other words, the constructed representation does not have to be an exact representation of all possible control flow, but only a precise enough, conservative estimate. When all matched patterns are unambiguous in this sense, the constructed graph of the program will be conservative and useful. Obviously, the degree of precision that is needed depends on the analyses and transformations one wants to apply.

When some code fragment cannot be matched to any unambiguous pattern, our link-time rewriter cannot build a program representation that is both conservative enough and precise enough to enable reliable

and useful rewriting. Whenever such a fragment is found, the rewriter therefore informs the developer of this fact. The developer then basically has four options. First, he can of course rewrite his program to the extent that it only includes matchable patterns. Obviously, requiring a developer to rewrite every program that contains unmatched fragments is not very user-friendly, let alone automated. Secondly, the developer can extend the set of patterns that are implemented in the link-time rewriter. Once a pattern is implemented, it can be reused for all programs to be rewritten. Thirdly, the developer can adapt the compiler, assembler, or linker in his tool chain to provide additional information on the generated code, that is used in the link-time rewriter. Again, the resulting tool chain can be reused for all of the developer's programs. Finally, the developer can instruct the link-time rewriter to ignore the code fragment in its optimizations, by treating it as immutable data that will not be rewritten, and making worst-case assumptions about the code fragment for all analyses. Obviously, this is only feasible if the code fragment is small enough so that the negative effect on the overall precision of the analyses is limited. In practice, the best approach is to combine the second and third option. To enable link-time rewriting of a program as unconventional as the Linux kernel, we have gradually implemented additional patterns and additional information provided by the compiler and linker, as discussed in Chapter 3.

As a result, there are currently only two (small) unmatchable patterns in the Linux kernel, and none in any other program in our extensive regression test suite, which consists of tens of programs compiled for multiple target architectures and run-time environments. One of the most important reasons is that, under separate compilation, code from one source code module, be it compiled code or hand-written code, cannot refer to code in other modules without a description of at least some aspects of the reference through symbol and relocation information. In practice, this implies that all uses of, e.g., program counters in PIC, will only impact small pieces of code. By treating that code as constant data, that will not be rewritten, the remaining code can still be transformed. We do this for some parts of the kernel, as described in Section 3.1.

2.5 Established Compaction Techniques

In this section, we present some established link-time compaction techniques. Applying these techniques to the OS kernel will be the first step in our memory footprint reduction scheme.

Unreachable code elimination is the simplest compaction technique that can be applied on the AWPCFG, by iteratively traversing reachable code in the WPCFG part of the AWPCFG. To obtain good results, this optimization needs to be performed context-sensitively such that only realizable execution paths are considered in which calls match returns. To eliminate inaccessible data from the AWPCFG as well, it suffices to apply a slightly adapted reachability analysis on the AWPCFG. In this adapted version, control flow edges coming from the unknown node are only traversed after their corresponding data reachability edges were traversed. A more advanced version of this analysis was published by De Sutter et al. [DeSu01].

Besides unreachable code and data elimination, a number of more advanced control flow optimizations can be applied as well. These include *duplicate code removal* [DeSu02], *inlining* of small procedures or procedures with a single call site and *branch forwarding*. The first of these detects whether multiple copies of a procedure or basic block are present in a program. If there are multiple identical procedures, all but one of them are eliminated, and calls to these procedures are replaced by calls to the one remaining copy. If there are identical basic blocks, they can be outlined into a new procedure. The original occurrences of the blocks are then replaced by calls to the new procedure. Note that, while duplicate code removal at procedure level does not incur run-time overhead, doing the same at basic block level does. New procedure calls have to be inserted to call the abstracted basic blocks, and some register spills may have to be inserted. In order to limit the performance impact of duplicate code removal at basic block level, it is advisable to use profile information to select only those blocks for factoring that are infrequently executed.

Next, there are a number of known data flow analyses and related optimizations that can be applied to the compacted graph. These include *conditional constant propagation* and *interprocedural liveness analysis* [Much97, Debr00, DeSu05, DeSu06]. Note that conditional constant propagation in particular is a powerful analysis: it not only allows us to find register values that are constant at a given program point over

all possible executions, but it also aids in the detection of unreachable code. During the analysis, conditional branches are handled intelligently: if the available information shows that a conditional branch will always be taken or not-taken, the propagation only follows the relevant code path. Code paths over which no information was ever propagated can be considered unreachable and can be eliminated from the program.

As mentioned in Section 2.3, these data flow analyses analyze the use of registers as if they were global variables. In general, no analysis information is propagated about/through memory locations. There are, however, a number of exceptions to this rule:

- Most importantly, the symbol information available in the object files allows us to determine for some procedures that they respect the calling conventions. If a symbol with global visibility is attached to a procedure, that means the procedure is exported and can be called by code from other compilation units. As the compiler cannot know all calling contexts for such global procedures, it has to generate code that abides to the calling conventions. Hence, we can deduce that the callee-saved registers will remain unchanged over calls to global procedures, even though these registers may be saved to and restored from the stack within the called procedure. Note that this symbol information is optional. It is not needed to detect procedures correctly, but only to derive additional information on their behavior.

- When constant propagation is able to determine that some load instruction accesses a fixed memory location in a read-only data section[2], the data at that address can be propagated into the program.

- The constant propagation implementation for the i386 architecture in our link-time binary rewriter is capable of identifying the stack operations used for setting up function parameters and retrieving them. This knowledge is exploited within the constant propagation analysis to propagate function arguments into called functions, thus increasing the analysis precision. This modification to the constant propagation is described in detail by Van Put

[2]This is possible because statically-allocated addresses are propagated just like other numerical constants.

et al. [VanP07b]. A more general approach, that allows for constant propagation throughout a procedure's full stack frame, is proposed by Schwarz et al. [Schw01], but this technique is not implemented in our link-time rewriter.

- The *load-store forwarding* optimization removes redundant loads and stores from the code. For example, if two consecutive stores to the same address occur without an intervening load, the first store can be eliminated. Another optimization opportunity arises when a value is stored to memory and subsequently loaded back without intervening stores to the same address. In this case, the load instruction can be replaced with a register copy operation if the value is still available in some processor register.

- Through a simple local stack analysis, implemented as a peephole optimization, redundant push and pop instruction sequences within a single basic block can be removed. Such redundant instructions occur even within a single basic block because other link-time transformations have made them redundant. For example, if a small (one-block) procedure is inlined into its caller, it is possible to merge the call site, the inlined procedure and the return site into one basic block. On the i386 architecture, where procedure parameters are passed through the stack, the aforementioned local stack analysis, combined with register renaming, can eliminate the stack manipulations that set up the procedure parameters and clean up the stack upon return, and instead pass the parameters to the inlined procedure code directly in registers.

All mentioned techniques were previously studied in many user-space contexts [Muth01, Schw01, DeSu01, DeSu02, Debr00, Mado04, DeSu05, DeSu07]. In order to apply them to a more complex, unconventional program such as a kernel, some special precautions need to be taken, which are discussed in the next chapter.

Chapter 3

Challenges in Rewriting an Operating System Kernel

In most previous link-time rewriting research, a number of assumptions about the code to be rewritten are made. For example, it is often assumed that only a limited number of computations take place on code addresses, and that these computations are annotated with relocation information. While such assumptions often hold for conventional, compiler-generated code, they do not necessarily hold for manually written assembler code.

As the lowest layer in the software stack of an embedded system, the operating system kernel needs to work directly with the hardware devices. As such, the kernel needs to perform many operations that are not easily described in higher-level programming languages. Consequently, the kernel contains a lot of manually written assembler code.

This chapter presents an overview of the unconventional behavior of that assembler code and of other OS kernel peculiarities that we have encountered in the Linux 2.4 kernel for the ARM and i386 architectures. For each peculiarity, we describe the countermeasures that need to be taken to handle the kernel code conservatively during link-time rewriting, yet allow aggressive compaction. As the standard link-time optimizations, described in the previous chapter, can now be applied to the kernel, we will evaluate their impact on the kernel's memory footprint and performance.

3.1 Two Address Spaces

The operating system kernel begins execution in a very early stage of the system boot process, when the booting system is not yet fully initialized. On systems with virtual memory support, one of the remaining initialization tasks is to turn on the memory management unit (MMU) of the processor. Before this is done, all code runs in the physical address space. All code that is executed after the MMU is enabled runs in a virtual address space.

Ordinary linkers typically do not support two different address spaces in the same program. This problem is circumvented by the Linux kernel developers with some clever manual assembler programming. In particular, the pre-MMU code is written in assembler, and all addresses appearing in this code are manipulated to trick the linker into producing the correct physical addresses. This is possible because both the physical and the virtual addresses of the pre-MMU code are known beforehand, so the difference between the virtual address of an instruction as assigned by the linker and the physical address at which it will be executed is a known constant. The kernel developers then explicitly subtract this offset from the linker-generated addresses whenever an absolute physical address is needed. This trickery exploits a deep knowledge of internals of the linker being used (the standard GNU linker ld), and of its simplicity or, in other words, of its lack of complex analyses and transformations.

Unlike the simple GNU linker, a link-time program rewriter is not limited to relocating addresses in the generated executable. Instead, it will also try to optimize the address computations. Consequently, the assembler code manipulations used to trick the standard linker into generating the correct addresses for this pre-MMU code will no longer work. Instead, they will confuse a standard link-time optimizer and result in faulty optimization of the address computations. To circumvent this, countermeasures need to be taken.

Fortunately, the amount of code that is executed in the physical address space is small compared to the other code. For example, on the ARM platform it is only 540 bytes large. Moreover, the code executed in the physical address space is easily identifiable, as all of this code is defined in one source code file (`arch/arm/kernel/head.S`). As such, the simplest way to deal with this problem is to exclude this code from all optimizations by simply treating it as a data section in the AWPCFG.

Because of the relatively small amount of code involved, the negative impact on the obtained compaction results is negligible.

3.2 Initial Page Tables

The memory management unit stores the virtual-to-physical address mappings in page tables, which are blocks of memory, typically 4 KiB in size. The initial page tables, which are used during the first stage of booting, are statically allocated in the Linux kernel. On the i386 platform, these initial page tables are stored as a data block within the kernel's code section.

While link-time rewriters usually know how to deal with data blocks occurring in the code section (this is a very common situation on several architectures, e.g., the ARM), these page tables introduce several problems. Firstly, link-time rewriters typically assume that data blocks in code sections can be placed more or less randomly, without any special alignment requirements. This is not the case here, as the page tables have to be aligned at a 4 KiB boundary. If they are not aligned correctly, the system will crash. We have solved this problem by replacing the standard layout algorithm of our link-time rewriter with one that first places the initial page tables at the correct address, and only then places all other code and data around them. Secondly, link-time rewriters traditionally consider the code sections of a program to be read-only (assuming the program does not contain self-modifying code), and as such the constant propagation analysis in our link-time rewriter will propagate values produced by constant loads in the code sections into the program. The initial page tables are not read-only, even though they appear in the code section. Consequently, constant propagation will produce faulty results when loads from the page tables are encountered. Therefore, we have modified our link-time rewriter's constant propagation algorithm to disregard loads from the initial page tables.

3.3 Initialization Code and Data

During the boot process, the kernel sets up the system and initializes a number of data structures and hardware devices. Most of the code and data structures used during this initialization process become useless

afterwards. But unless countermeasures are taken, they keep occupying memory.

To avoid this, the Linux kernel developers annotate such initialization code and data, and instruct the compiler and linker to put them into separate code and data sections, the so-called *init sections*. Once all initialization is done, the kernel releases the virtual memory pages on which the init sections reside, thus freeing the memory they occupied.

During the analysis and optimization phase of the link-time rewriter, all code sections of the kernel are joined in a single AWPCFG, and compacted as a whole. Compaction techniques such as code factoring and branch elimination can make it unclear whether a basic block should belong to the initialization sections or not. During the layout phase, when the control flow graph is transformed into a linear representation, the link-time rewriter must therefore decide which code belongs in the init sections. It is important that no code ends up in the init sections by mistake, as that would mean it disappears from memory after initialization, while it may still be needed afterwards. On the other hand, the optimizations should not cause too much code to be transferred from the init sections to the regular code section. Doing this may result in a smaller overall code size (which is good if optimizing the static ROM footprint is the goal), but it would also result in a larger resident code size after initialization (which is bad if optimizing the RAM footprint of the kernel is the goal).

Our link-time rewriter marks all AWPCFG nodes that originally came from init sections. The markings have to be kept up to date: if duplicate code removal merges two code fragments, the resulting procedure may only be marked as init code if all of the original code fragments were init code as well. In general, to achieve the best RAM footprint reduction, the link-time optimizations should prioritize size gains in the non-initialization sections over size gains in the initialization sections. For example, if duplicate code removal discovers n identical basic blocks, of which $n - 1$ are initialization code, it will exclude the one non-initialization basic block from the factoring. Otherwise, the factored-out function would have to be placed in the non-init sections, so the non-initialization code size would increase, even though the total code size decreases. If the optimization goal is reducing the kernel's ROM footprint, the link-time optimizer should clearly not make the same prioritization. In that case, it is better to reduce the kernel's overall code size, at the cost of more RAM usage afterwards. In the cases

where such a decision had to be made, we have chosen to optimize for RAM footprint over ROM footprint. In the layout phase, our rewriter takes care to place the marked basic blocks back in the init code section.

3.4 Manually Written Assembler Code

Besides the physical address space initialization code, there are numerous other occurrences of manually written assembler code in the kernel. Procedures written in assembler code do not always adhere to the calling conventions or the ABI of the target platform, even though they may be exported to other source code modules. This is the case when all call sites of a manually written assembler procedure are written in assembler as well. In such cases, the kernel developers have full control over the parameter passing mechanism that they want to impose. When such developer-imposed conventions differ from the standard conventions, the involved, exported procedures violate the assumption put forth in Section 2.5 that exported procedures always respect the architecture's calling conventions.

In theory, there are three ways to treat such unconventional assembler code conservatively. The simplest option is to neglect the existence of calling conventions altogether. If no program analysis assumes that calling conventions are maintained, no analysis will produce incorrect results where the conventions are not maintained. However, this option is not viable, because there are many cases in which assumptions about the calling conventions do yield useful information, such as with the propagation of data flow information of callee-saved registers, as mentioned in Section 2.5.

The second option to deal with code that does not maintain calling conventions consists of using code inspection to detect such code. The detected fragments can then be differentiated from conventional code in all program analyses. We do not find this to be a viable option either, because detecting whether or not a procedure's stack behavior respects the calling conventions would be either very complex (due to the problems of aliasing memory accesses [Debr98]) or too imprecise [Linn].

This leaves us with the third option, in which the compiler informs the link-time rewriter of all manually written assembler code. This requires patching the compiler tool chain with which the kernel is compiled. Fortunately the required patch is extremely simple.

For the GCC tool chain, for example, a 3-line patch to GCC's `specs` file (that specifies the configuration of the tool chain) forced the GNU compiler to add two mapping symbols to the generated object code for each piece of inline assembler code. In particular, a label `$handwritten` is now added at the beginning of all inline assembler code fragments in the generated object files, and a label `$compiler-generated` is added at the end of each inline assembler fragment. These mapping symbols are a common concept in tool chains. For example, the ARM ABI requires compilers to add mapping symbols to indicate which parts of the code section contain ARM code (`$a`), Thumb code (`$t`), or data (`$d`). These mapping symbols are needed by the linker to produce a correct executable.

Furthermore, each object file produced by the GCC tool chain for ELF targets contains the name of the source file it was generated for. Hence full assembler files (such as `head.S`) can be detected at link-time by looking at the extension of the source code file name (".S" or ".s").

During the link-time rewriting of the kernel, each procedure of which the labels or source file name indicate that it contains manually written assembler code, and of which all call sites are also written directly in assembler code, is treated as an unconventional procedure, i.e., a procedure not respecting the calling conventions. Note that these unconventional procedures can still be analyzed and optimized, the link-time rewriter just assumes they do not adhere to the calling conventions, and hence computes less precise, but still conservative, data flow information on them.

In the Linux kernel configured for our ARM test platform this solution allows us to assume that 1720 out of 3925 procedures respect the calling conventions. Only 9 global procedures need to be treated as unconventional because they are written in and called from hand-written assembler code. For our i386 test platform we can assume adherence to the calling conventions for 1717 out of 4939 procedures, and 120 global procedures are treated as unconventional on account of being written in and called from hand-written assembler code.

3.5 Memory-mapped Input/Output

In many cases, the kernel communicates with peripheral devices by means of memory-mapped I/O: by writing to or reading from special memory locations the kernel can issue commands to or read data from

these devices. Obviously, memory accesses to these memory-mapped I/O addresses have very different properties from accesses to regular memory. For example, two successive reads from the same memory location without an intervening write are usually expected to return the same value. For memory-mapped I/O addresses this is not necessarily the case. This means that optimizations like load-store forwarding that reorder or even remove memory operations are not allowed on memory accesses that implement memory-mapped I/O.

In practice, it is virtually impossible to distinguish regular memory accesses from I/O memory accesses at link time. Only memory accesses relative to the stack pointer (or to registers whose value was derived from the stack pointer) are guaranteed to be regular memory accesses. Consequently, we modified our link-time rewriter to only perform memory access related optimizations on stack accesses.

3.6 Special Instruction Sequences

Besides special privileged mode instructions that do not occur in user-space applications, the Linux kernel contains some sequences of seemingly innocuous instructions that require special treatment. Usually, these sequences depend on the micro-architectural side-effects of instructions to influence the processor operation on a level that is normally hidden from the application programmer. For the i386 Linux 2.4 kernel there are two such sequences:

- Writing to the processor's control registers:

```
                mov %cr0, %eax
                orl $0x80000000, %eax
                mov %eax, %cr0
                jmp <next>
<next>:         ...
```

  This instruction sequence enables the processor's memory management unit, switching the execution context from physical to virtual address mode. The first three instructions set the paging bit in the appropriate control register. The jump instruction flushes the processor's prefetch queue to ensure all subsequent instructions are interpreted in the new processor context. This jump does not alter control flow and only appears in the sequence

because of its side-effect. A regular link-time optimizer would not take this side-effect into account and would remove the jump from the program. We modified our link-time rewriting framework to let it check whether or not such a seemingly useless jump is preceded by an instruction that alters a control register. If this is the case, the jump instruction is marked as having a side-effect and will never be removed.

- The `BUG` sequence:

```
ud2
<source line number encoded as 2-byte int>
<pointer to source code file name>
```

  This sequence is used to signal bugs in the kernel code. The `ud2` instruction causes an "undefined instruction" exception. The exception handler then uses the address of the instruction that causes the fault to locate the source code file and line number information that is printed on the console, after which execution is terminated.

  There are no explicit references to the source code information immediately following the `ud2` instruction, so normally the link-time rewriter would consider this data to be unreachable and remove it from the kernel. We have adapted our link-time rewriter so that it adds a data reference edge from the `ud2` instruction to the source code data. As long as the instruction is reachable, the data will remain reachable as well. During the control flow graph linearization phase, special care is taken to place the data immediately after the `ud2` instruction.

For the ARM Linux kernel, there are three special sequences:

- The `cpwait` instruction sequence:

```
mrc p15, 0, r1, c2, c0, 0
mov r1, r1
sub pc, pc, #4
```

  This instruction sequence makes sure that, after a write to the system control coprocessor (the equivalent of the i386's control registers), all following instructions are interpreted according to

the new processor state. The first instruction reads some random value from the coprocessor and the second instruction forces the processor to stall until this read is completed, thus ensuring that the coprocessor write instruction is completed before execution continues. In other contexts, the second instruction would be useless as it does not change the visible processor state. The third instruction executes a jump to the next instruction, which flushes the processor pipeline. Again, this instruction would normally be considered useless as it does not alter the control flow.

- The `cpwait_ret` sequence combines `cpwait` with a function return:

```
mrc p15, 0, r1, c2, c0, 0
sub pc, lr, r1, lsr #32
```

The `mov` and `sub` instructions from the previous sequence are now combined into a single instruction that subtracts 0 from the link register `lr` (which holds the return address) and stores the result in the program counter `pc`. A good link-time optimizer would note that shifting a 32-bit value right over 32 positions results in 0 and would replace the `sub` instruction with `mov pc, lr`. By removing the dependency of this instruction on `r1`, the processor would no longer need to stall, resulting in potential execution errors.

- D-cache initialization on the PXA250 processor:

```
          bic r0, pc, #0x1f
          add r1, r0, CACHESIZE
<loop>:   ldr r2, [r0], #32
          cmp r0, r1
          bne <loop>
```

The first instruction copies the program counter in `r0` and zeroes the lowest five bits. Subsequently a loop is executed that loads some values into `r2` that will never be used, which makes the load instruction in this loop a prime candidate for elimination after register liveness analysis is performed. The real purpose of this loop is to initialize the data cache on the processor, and this effect would be lost if the load instruction were eliminated from the kernel. While it may seem useless to initialize the data cache

by filling it with random data from the code section, this code is necessary as a workaround for a cache bug in the PXA250 processor.

Again, we have modified our link-time optimizer to recognize these sequences and mark them as having side effects, thus ensuring that they will not be altered during the optimizations.

3.7 Exception Handling

Apart from the explicit control flow through jump and call instructions, there is also implicit control flow caused by processor exceptions. For user space programs, processor exceptions (e.g., page fault exceptions) are handled transparently by the OS kernel, and as such they can safely be ignored by link-time rewriters. When an exception occurs within the kernel code itself, it has to be handled explicitly, which implies that our link-time rewriter has to take it into account.

3.7.1 Exception Handling in the Linux Kernel

The Linux kernel contains a list of all instructions of which the developers expect they can raise exceptions. Mostly, this concerns loads from or stores to user space memory, as page faults can occur upon execution of the load or store. Memory accesses to kernel memory will never cause page faults as the kernel memory is never swapped out. This list is stored in the `__ex_table` section, and it can be perceived as a two-column list: the first column contains the address of the instruction that possibly raises an exception, the second column stores a pointer to the so-called *fixup code* for this instruction. The fixup code carries out the appropriate actions in case an exception occurs, like for example setting an error code in a register.

The kernel contains a generic exception handler that is invoked whenever a processor exception arises. The handler then looks up the address of the faulting instruction in the `__ex_table` section and transfers control to the appropriate fixup code. If the faulting instruction is not listed in the table, an unexpected situation has occurred and the kernel aborts execution with an error message. If the generic handler passes control to a fixup code fragment, it takes care to restore the original processor context first, so to the executing thread it appears as if

control has transferred transparently from the faulting instruction to the fixup code. This allows us to leave the generic handler out of the control flow modeling without introducing inaccuracies in the graph.

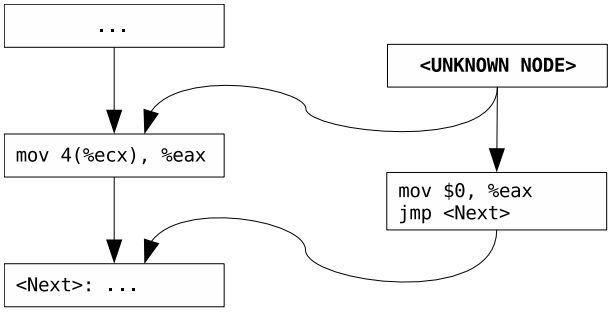
3.7.2 Challenges for a Link-time Rewriter

This approach to exception handling brings about several problems for a link-time binary rewriter:

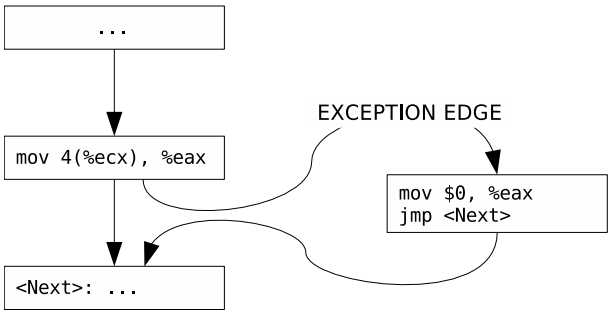
- Using only the rules described in Section 2.3, the link-time rewriter would model the exceptional control flow incorrectly. Figure 3.1 illustrates this. In part (a) we see the naïve modeling of the exceptional control flow. Both the instruction that can potentially raise an exception and the fixup code have an incoming control flow edge from the unknown node because both are referred to from the `__ex_table` section. This is overly conservative: the references in the first column of the exception table will never be used for indirect control flow and as such the corresponding control flow edge is superfluous. We also see that there is no indication in the control flow graph that control can branch from the faulting instruction to the fixup code. This is definitely incorrect, as it violates the rule that a conservative control flow graph always represents a superset of all possible execution paths. Part (b) of the figure shows the ideal modeling of the control flow. For this, a special type of control flow edge is introduced, the *exception edge*, that directly connects the potentially faulting instruction with the corresponding fixup code. In this representation, which was proposed by Rajagopalan et al. [Raja06], there is no need for the unknown node, and as such it is very precise. As our link-time rewriter does not support this concept of exception edges, we have opted for a more conservative, but still correct, modeling, which is shown in part (c) of the figure. The exception edge is replaced with a control flow edge to the unknown node, and the fixup code keeps its incoming edge from the unknown node.

- Analyses and optimizations need to be aware of the exception handling situation. For example, duplicate code removal should not factor out two identical-looking procedures that have non-equivalent fixup fragments associated with them.

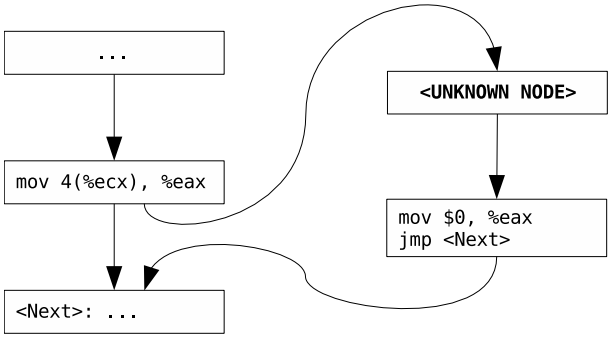
- Whenever an exception arises, the execution of the instruction



(a) naïve modeling



(b) ideal modeling



(c) conservative modeling

Figure 3.1: Control flow graph modeling of the exception handling mechanism of the Linux kernel.

that caused the exception is canceled. This is something that analyses should take into account: when propagating information over an exceptional control flow path, they should consider the faulting instruction to be unexecuted. When propagating information over the regular execution path, the instruction has to be considered executed. This is mainly important for link-time binary rewriters that use the ideal modeling of exceptional control flow. The modeling we used is not susceptible to this problem, as the control flow path leading to the fixup code passes through the unknown node, which means that analyses have to assume worst-case properties for this path, and all information propagated over this path is lost.

- The generic exception handler looks up the address of the faulting instruction in the `__ex_table` section through binary search. This means the entries in the section have to be in sorted order. There is however no guarantee that the kernel code will be laid out in the original order after the rewriting is done. Therefore we have modified our rewriter so that it sorts the entries in the `__ex_table` section after the graph linearization phase of the rewriting process is completed.

3.8 Evaluation

Once the link-time binary rewriter is adapted to support all aforementioned kernel code peculiarities, it can be used to apply the standard link-time optimizations described in Section 2.5 to an OS kernel. In this dissertation we will use two test systems to perform the evaluation of all proposed techniques. In this section, we will first describe the two test configurations in detail, and then we will evaluate the impact of the standard link-time optimizations on the memory footprint and performance of the kernel.

3.8.1 Evaluation Environment

For the evaluation, we have used two different Linux systems, one based on the i386 architecture, the other one based on the ARM architecture.

Our i386 system has a Pentium III processor, with 64 MiB of RAM,

an IDE hard disk and a Fast Ethernet network card. The Linux kernel is a vanilla 2.4.25 kernel, configured without module support and with only the necessary drivers for this system. Compilation was done with GCC 3.3.2.

Our ARM system is an Intrynsic CerfCube 255, with a PXA255 XScale processor, with 64 MiB of RAM, 32 MiB of Flash storage and a Fast Ethernet network connector. The Linux kernel is a 2.4.19 kernel, with patches supplied by the device manufacturer. The kernel is also configured without module support and with only the necessary drivers. Compilation was done with GCC 3.2.

For both kernels, the compiler was instructed to optimize for code size (optimization flag `-Os`). All other build options were left at their standard values.

Both systems have an identical user space configuration that approximates the firmware of an internet home gateway (e.g., the Linksys EtherFast Cable/DSL router), with native address translation (NAT) features enabled in the kernel configuration and a dynamic host configuration protocol (DHCP) client (for acquiring an IP address from the Internet service provider) and server (for assigning IP addresses to the systems on the local network) and a simple web server to allow for browser-based configuration of the device.

These kinds of devices commonly use Linux-based firmware, so this is a realistic case study. The user space software is based on Busybox 1.4.2[1]. This is a so-called multicall program that performs different functions depending on the name by which it is called. It is used in almost all embedded Linux systems because it provides a very compact, but complete user space environment. On our test systems, Busybox acts as `init`, as a shell, as a number of other necessary system utilities and even as the web server and DHCP client and server. Busybox is statically linked against uClibc[2], an embedded C library that is also engineered for small code size.

To assess the performance impact of the applied transformations on the resulting kernel, we use LMbench 2.0.4 [McVo96], a cross-platform benchmark suite for Unix-like operating systems that measures various aspects of the kernel's performance, like system call performance, interprocess communication bandwidths and latencies and context switching times.

[1] `http://www.busybox.net/`
[2] `http://www.uclibc.org/`

3.8.2 Impact on Kernel Footprint

Table 3.1 shows the impact of the standard link-time optimizations on the kernel memory footprint for our two test systems. All sizes in the table are in kibibytes. In each row, the technique mentioned in the left-most column is applied in addition to the techniques mentioned in the rows above. The bottom two rows show results with restricted duplicate basic block elimination. These will be discussed later on. Columns two and three present the sizes of the non-initialization code and data sections of the kernel. The data size includes the read-only, writable and zero-initialized data sections. The next two columns present the sizes of the initialization code and data sections. The next column shows the size of the resulting `vmlinux` image (the uncompressed kernel image), in which the zero-initialized sections occupy no space. The seventh column shows the size of the corresponding compressed kernel image file. The last two columns show the kernel's static RAM footprint, respectively during and after system initialization.

The inclusion of both uncompressed and compressed image sizes in the table merits some further explanation. In the first instance, the output of the Linux build process is an uncompressed kernel image called `vmlinux`. This is a regular ELF executable file, just like a user space program. It is on this representation that the link-time optimizer will apply its optimizations. On some systems, particularly those that use execute-in-place (XIP) technology to execute the kernel directly from Flash memory, the `vmlinux` image is placed in ROM and as such the size of this image is equal to the kernel's static ROM footprint on these systems. However, it is customary to append another step to the Linux build process, in which the kernel is transformed into a self-extracting compressed image, called `zImage` or `bzImage` depending on the target architecture. The compression is performed with the well-known `gzip`[3] compressor, and a small decompressor is attached to the compressed image that will reconstruct the original `vmlinux` image in memory at boot time. In this way, the kernel's static ROM footprint can be significantly reduced, at the cost of precluding XIP techniques. For systems where this extra compression step is used (which is the case for most current-day systems), the static ROM footprint is shown in the compressed image size column of the table.

[3] `http://www.ietf.org/rfc/rfc1951.txt`

	text size	data size	init text size	init data size	image size	compressed image size	init mem. footprint	memory footprint
original kernel	714	292	46	8	932	470	1061	1006
Compaction techniques								
+unreachable code/data elim.	678 *-5.0%*	273 *-6.6%*	46 *-1.3%*	8 *-2.2%*	871 *-6.5%*	456 *-3.0%*	1004 *-5.3%*	951 *-5.5%*
+duplicate code removal	635 *-11.1%*	264 *-9.8%*	45 *-2.0%*	8 *-2.2%*	819 *-12.1%*	458 *-2.5%*	952 *-10.3%*	899 *-10.7%*
+whole-program optimization	632 *-11.6%*	263 *-9.9%*	45 *-2.4%*	8 *-2.2%*	815 *-12.5%*	456 *-2.9%*	948 *-10.6%*	895 *-11.1%*
Restricted duplicate basic block elimination								
+profile information	632 *-11.5%*	263 *-9.9%*	45 *-2.4%*	8 *-2.2%*	815 *-12.5%*	456 *-3.0%*	948 *-10.6%*	895 *-11.0%*
- duplicate basic block elim.	641 *-10.3%*	263 *-9.9%*	45 *-1.7%*	8 *-2.2%*	827 *-11.2%*	448 *-4.7%*	957 *-9.7%*	904 *-10.1%*

(a) i386

	text size	data size	init text size	init data size	image size	compressed image size	init mem. footprint	memory footprint
original kernel	1060	203	47	4	1207	575	1315	1264
Compaction techniques								
+unreachable code/data elim.	996 *-6.1%*	203 *-0.2%*	46 *-2.3%*	4 *0.0%*	1105 *-8.4%*	576 *0.3%*	1249 *-5.0%*	1199 *-5.1%*
+duplicate code removal	954 *-10.0%*	203 *-0.2%*	45 *-4.4%*	4 *0.0%*	1065 *-11.7%*	576 *0.2%*	1206 *-8.2%*	1157 *-8.4%*
+whole-program optimization	935 *-11.8%*	203 *-0.2%*	44 *-7.2%*	4 *0.0%*	1041 *-13.7%*	574 *-0.1%*	1185 *-9.8%*	1138 *-10.0%*
Restricted duplicate basic block elimination								
+profile information	936 *-11.7%*	203 *-0.2%*	44 *-7.2%*	4 *0.0%*	1041 *-13.7%*	574 *-0.1%*	1186 *-9.7%*	1139 *-9.9%*
- duplicate basic block elim.	965 *-9.0%*	203 *-0.2%*	45 *-5.8%*	4 *0.0%*	1073 *-11.1%*	572 *-0.5%*	1216 *-7.5%*	1168 *-7.6%*

(b) ARM

Table 3.1: Impact of the standard link-time optimizations on the kernel's memory footprint.

General Discussion

Applying well-known, general purpose link-time compaction techniques to the kernel results in a reduction of the static RAM footprint after initialization of about 10% on the ARM and 11.1% on the i386 platform. Approximately half of this gain can be attributed to unreachable code and data elimination. Given the size of the Linux kernel source base this does not surprise us. Any large project that has evolved over a long period of time is bound to contain unreachable code and inaccessible data. It does however indicate that there is still some margin for improving the kernel configurability, through which unused code should normally be excluded.

It should be noted that some of the size reductions obtained with unreachable code and data elimination can in theory also be achieved by simply compiling the kernel with the compiler optimization flags `-ffunction-sections` and `-fdata-sections`, and linking the resulting object files with the `--gc-sections` flag. These flags instruct the compiler to place every procedure and every global variable in its own section. The linker can then remove all unreferenced sections from the final binary. This is somewhat similar to our link-time compactor's unreachable code elimination, albeit at a coarser granularity. Moreover, invoking these compiler flags deteriorates the quality of the generated code, as the compiler can perform less address computation optimizations. A true link-time optimizer obviously does not suffer from this drawback.

Most of the other gain, especially on the i386, comes from duplicate code removal. Duplicate code removal gains at the procedure level result from the fact that there are a lot of similar procedures in the kernel, that operate on superficially different data structures (e.g., a list of pointers to virtual memory pages versus a list of pointers to open files), from cut-and-paste duplication by the developers, and from the fact that the GCC compiler does not always honor the inlining requests of the programmer. This last cause merits some more discussion. In the Linux kernel, a number of procedures are defined in header files as `static inline`. If these are inlined properly, the compiler optimizations are able to remove large parts of these procedures through specialization because the exact calling context for each inlined instance is known. The GCC compiler does not always perform this inlining however, and as a result these procedures appear several times throughout the kernel code in their original, non-specialized form.

Duplicate code removal gains at the basic block level result from similar procedures in the kernel that differ enough to be unsuitable for procedure-level factoring, but have a number of basic blocks that are identical. The other major source of opportunities for basic block factoring comes from the use of inline assembler macros in the source code. These macros, that implement things like copying a value from user space memory to kernel space, appear quite frequently, and are always inlined into their callers.

The other whole-program compaction techniques implemented in Diablo add another 1.6% to the obtained compaction for the ARM, but amount to practically nothing for the i386. This is because most of the analyses treat memory as a black box. As the i386 architecture lacks sufficient user-visible registers, almost all computations involve the stack. This makes the data flow analyses very imprecise.

It may seem remarkable that none of the transformations have an impact on the size of the data sections for the ARM kernel, while they are capable of removing almost 10% of the same data on the i386. The reason is simple however: all read-only data is incorporated into the code section on the ARM, and is thus counted as code. On the i386 this data resides in a separate section and is counted as data. The compaction techniques typically have much more impact on the read-only data sections than on mutable data sections, which explains the very small impact of the compaction techniques on the ARM data sections.

Impact on Compressed Image Size

It is clear that the gains obtained for the compressed image size are somewhat disappointing. For the i386 architecture, unreachable code and data elimination reduces the size of the compressed image by 3%, but duplicate code elimination reduces this gain to 2.5% even though the size gain for the uncompressed image nearly doubles from 6.5% to 12.1%. For the ARM architecture, the results are even stranger: despite reducing the uncompressed image size with 8.4%, unreachable code and data elimination slightly increases the compressed image size. Subsequent optimizations offer no substantial improvement: the total compressed image size reduction amounts to 0.1%.

To some extent the difference between uncompressed image size reduction and compressed image size reduction is understandable. Gzipping, like any compression technique, reduces the amount of redun-

dant information in a byte stream. Compaction techniques such as duplicate code removal also remove redundancy from a program, albeit on a different level. In fact, duplicate basic block removal even decreases the compressibility of the remaining code by replacing easily compressible information (identical pieces of code) by information that is much harder to compress (calls to the extracted procedure, which all contain different relative displacements). Duplicate code removal at the procedure level does not have the same detrimental effect, as it does not involve the insertion of additional, hard-to-compress function call instructions in the program code. Instead, duplicate procedure elimination only replaces calls to one function by calls to another function.

To quantify the effect of duplicate basic block elimination on the compressibility of the kernel image, we applied all compaction techniques, minus the duplicate basic block elimination. The resulting total compaction is presented on the last row of Table 3.1(a) and (b). These results confirm our argumentation, as disabling duplicate basic block removal improves the compressed image size by 1.8% on the i386 and by 0.4% on the ARM, even though the uncompressed image sizes increase significantly.

The viability of duplicate basic block elimination hence depends on one's optimization target. When the goal is reducing the static RAM footprint, this optimization should certainly be applied. If the goal is reducing the compressed image's size however, this transformation is best disabled.

While the effect of duplicate basic block elimination completely accounts for the increase of the compressed image size for the i386 test system, this is clearly not the case for the ARM system. Here, the gzipped image already grows when the unreachable code and data is removed from the kernel, notwithstanding the fact that the corresponding uncompressed image was reduced with 8.4%. The only difference between this kernel image and the original one is that the unreachable code and data have disappeared from the image and that the code and data layout has changed. To understand this result, we have examined a large number of regular ARM programs (from the SPEC and MediaBench benchmark suites), on which we applied several different code and data layout algorithms before gzipping them. Although we always observed a similar behavior, we have until this day not been able to pinpoint precise causes of this behavior, as we have not observed any systematic relation between code layout properties (such as average

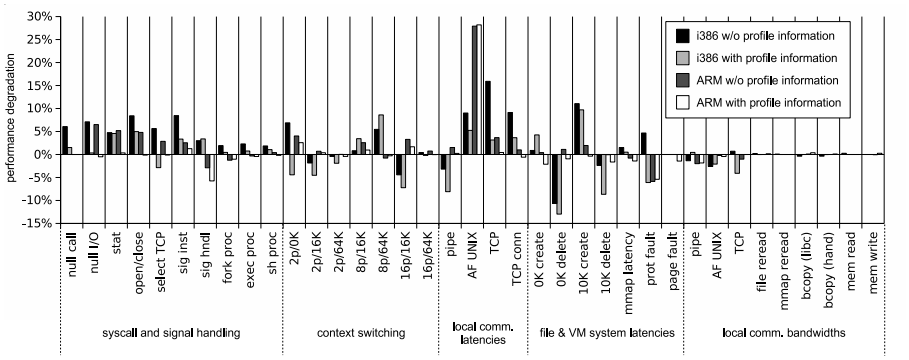


Figure 3.2: Performance degradation for the LMbench benchmark suite.

branch displacement) and compressibility. Consequently, this remains an open, and in our opinion, very intriguing question.

3.8.3 Impact on Kernel Performance

As the focus of our research is memory footprint reduction, our link-time rewriter does not include optimizations specifically targeted at execution speed improvement. Therefore, we do not expect significant speedups in the rewritten kernels. However, we also want to avoid excessive slowdowns caused by the applied optimizations. In theory, the only applied optimization that can cause significant slowdowns is the duplicate basic block removal, as this transformation introduces extra function calls and potentially extra register spills and restores in the kernel. However, it is possible to limit the performance impact this optimization causes by using profile information to guide the basic block factoring process. Therefore, we have gathered such profile information, and we will evaluate in which way it impacts the kernel's execution speed and the attainable memory footprint reductions. How the profile information is collected will be discussed in Appendix A.

The graph in Figure 3.2 shows the performance degradation of the rewritten kernels with respect to the original kernels. Positive bars correspond to slowdowns, whereas negative bars indicate speedups. For each microbenchmark we show results for the i386 and ARM kernels, with all optimizations enabled, with and without profile information to guide the duplicate basic block elimination.

Without profile information, the average performance degradation is 2.4% for the i386 kernel and 1.5% for the ARM kernel. With profile information, the i386 kernel experiences a marginal speedup of 0.1%, while the average slowdown of the ARM kernel is reduced to 0.3%. As can be seen in the next-to-last row of Table 3.1 (a) and (b), the restricted duplicate basic block elimination reduces the memory footprint gains with at most 0.1%. As such, it is clearly advisable to use profile information to guide the basic block factoring process, as this significantly reduces the performance impact without severely restricting the attainable memory footprint gains.

Chapter 4

Specialization for a Known System Configuration

Embedded devices often have known, fixed hardware and a fixed set of user space applications. Examples of such fixed-function systems are the Linksys WRT54GL wireless Internet gateway[1] and the TiVo digital TV recorder[2], both of which run the Linux kernel. For these systems, it is known a priori which kernel functionality is required, and which is not.

The built-in configuration capabilities of the Linux kernel allow the system designer to select the required hardware drivers semi-automatically. While this driver selection can be done at a very fine-grained level, it can only be used to omit drivers (and some other functionality) from being compiled and linked into the kernel image. In most cases, this built-in configuration does not allow the remaining, selected parts of the kernel to be optimized for a selected configuration. For example, even though there may be no need to provide boot-time command-line parameters for some driver on a particular system, it is not possible to omit the code for handling such command-line parameters automatically. Hence there is no automated method for optimizing the driver for its default parameter values, let alone for other, fixed values. While a user of a general-purpose computer might be interested in booting the kernel with different command-line parameters, this rarely is the case for embedded systems. On PDAs, mobile phones, and other such embedded systems, the user is most often not supposed to influ-

[1] `http://www.linksys.com/`

[2] `http://www.tivo.com/`

ence the boot process at all. Therefore, there is no reason to retain the command-line kernel configurability for such systems, or indeed to retain the overhead that results from the lost optimization opportunities. On embedded systems running Linux, manual techniques are typically used to remove this overhead.

With respect to the software needs, a fine-grained configuration is not at all available in the standard distribution of the Linux kernel. For example, most of the system calls implemented in Linux cannot be omitted with the standard build-time configuration, even though they may not be needed on specific systems. The system calls include, for example, different versions of calls that correspond to different versions of the standard GNU C library implementation. Such system calls are included for backward compatibility, but on a system with fixed software, including a fixed C library, it is perfectly well known which versions of such system calls are required. Furthermore, most embedded systems are not as general-purpose as the standard kernel, and hence embedded systems often do not require all the functionality that the kernel exposes to user space through system calls. It should be noted that specialized, commercially supported distributions of the Linux kernel exist (e.g., LynuxWorks Bluecat, Montavista Linux) that allow more fine-grained configuration. However, this configurability is introduced manually, which means it has to be reintroduced, or at least updated, for every new release of the Linux kernel these distributions support.

In this chapter, we propose and evaluate link-time kernel compaction and specialization optimizations based on a known hardware/software configuration and a fixed boot process. The major benefit of applying these techniques at link time is that they do not require the source code to be changed, and that the specialization they offer hence does not complicate the maintenance of the kernel source. Furthermore, as the kernel-specific specializations are applied at link time, they can cooperate seamlessly with the existing link-time program transformations discussed in Section 2.5. Again, we focus on the Linux 2.4 kernel, but variations on these techniques are applicable to other operating systems as well.

Note that the proposed techniques remain applicable for devices whose functionality may change during their lifetime due to firmware upgrades that add new features or refine current ones. Usually, such firmware upgrades replace all the software on the device at once, in-

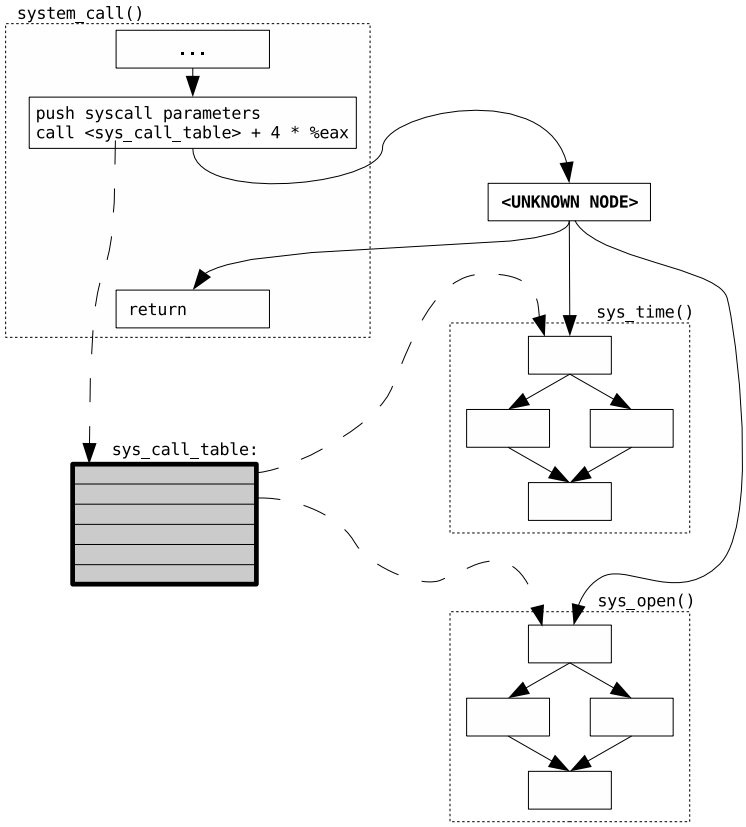


Figure 4.1: Kernel system call handling.

cluding the OS kernel. For each firmware version, an appropriate kernel specialized for the exact software configuration can be included in the firmware image.

4.1 System Call Elimination

The first kernel specialization technique concerns the removal of unused system call handlers. In Linux, all system calls are identified with an integer. Where a system call occurs in the code of a user-space application, this number is either encoded literally in the system call instruction or it is passed from user-space as the first parameter of the system call. The kernel then uses this number to index the system call handler

table, from which it loads the address of the corresponding handler, to which control is then transferred.

Figure 4.1 illustrates the system call handling mechanism of the i386 Linux kernel. When an application issues a system call, the processor invokes a generic system call handler in the kernel. This generic handler then delegates the work to dedicated system call handlers based on the system call number (which is passed in register `%eax`). This dedicated handler is found by using the system call number as an index into the system call handlers table.

Because each dedicated handler's relocatable address is stored in this table, the AWPCFG includes a data reachability edge from the table to the handler, and a corresponding control flow edge from the unknown node to the handler. These edges keep the handler reachable in the kernel, even if the system call handler might not be called from within the kernel itself. For all system calls that are not used by any user space program, we remove the data reachability edge and the corresponding control flow edge from the AWPCFG. As a result, if a handler is not reachable in any other way from within the kernel, it will become unreachable in the AWPCFG, and the unreachable code and data elimination discussed in Section 2.5 will eliminate it. If the handler is reachable in any other way, either because it is called directly or because its address is stored in some other data structure in the kernel, other edges in the AWPCFG will keep the handler reachable.

In short, the only additional feature needed to implement this specialization in a link-time kernel rewriter is the possibility to gather a list of unused system call numbers, to identify the table at the `sys_call_table` symbol, and to nullify the unused entries by removing the appropriate AWPCFG edges.

To collect the list of system calls that can be eliminated, one has to analyze all programs that will be installed on the embedded system. For architectures like the ARM, where the number of the system call is encoded literally into the system call instruction, this is trivial: it suffices to find all system call instructions and disassemble them to generate a list of reachable system calls. On an architecture like the i386, where the system call number is passed in a register, constant propagation is needed to determine the value of this register at each system call instruction. To be conservative, we need to assume that all system call handlers can be called as soon as the system call number of one system call cannot be determined by the constant propagation. In practice,

we have found this not to be a problem, as a basic link-time constant propagation of register values was able to resolve all system calls in a large number of benchmarks that were linked against two different C libraries (glibc and uClibc) for the i386.

If not all user space programs are known a priori, this automated specialization may still be useful. On many systems, the installed system libraries are known beforehand. When applications are only allowed to perform system calls through these libraries, for example for security reasons, it is sufficient to analyze only these libraries.

4.2 System Call Specialization

After the unused system call handlers have been removed, the remaining ones can be specialized for known parameter values. By performing a constant propagation analysis on all user space programs, it is possible to determine all possible values for some of the system call parameters. This collection of known parameter values can be done along with the detection of used system calls as discussed in the previous section.

In the link-time rewriter, the collected information is used to improve the precision of the conditional constant propagation. As illustrated in Figure 4.2, the additional information can be injected in constant propagation without adapting the analysis code. It suffices to temporarily wrap the system call handler in a new procedure that sets up the known system call parameters before performing the actual call to the handler. In the example shown in the figure, we were able to determine all values for two parameters of the `open` system call: the second parameter always takes value 1, the third parameter can be 3 or 5. In the wrapper function, there is a code path for each possible combination of the parameter values, and on each path the known parameters are set up and the handler is called. For unknown parameter values, a random register is pushed onto the stack, as the registers will always contain unknown values because the wrapper procedure is called from the unknown node.

System call specialization is mainly useful for removing argument validity checks from the system call handlers and for removing functionality from multiplexed system calls. A multiplexed system call is one that performs completely different actions depending on the value of one of its arguments, for example, the `ioctl` and `socketcall` system

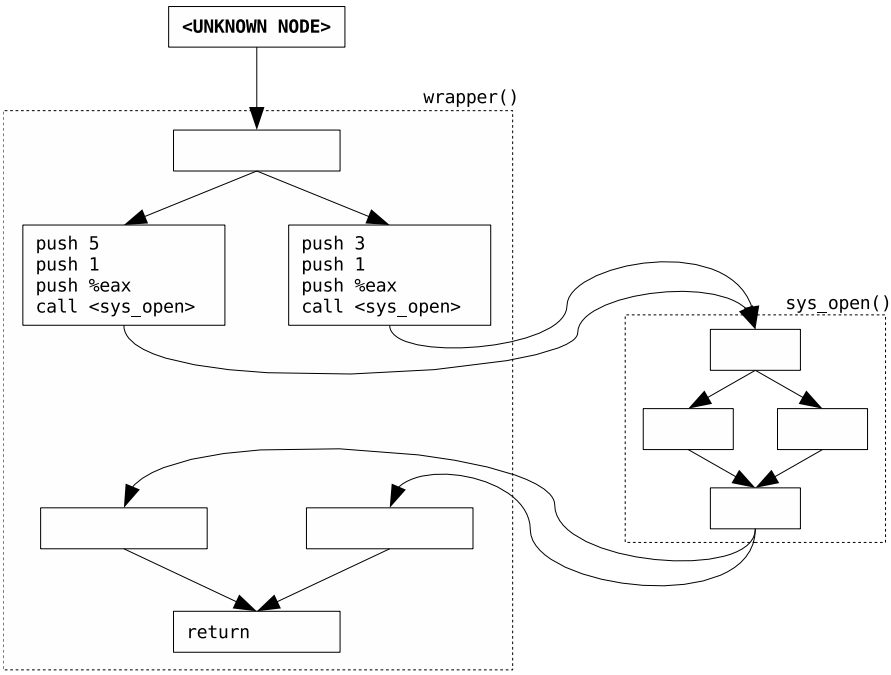


Figure 4.2: Information about constant system call parameters can be supplied to constant propagation through a wrapper procedure.

calls. If the value for the command argument is known for all invocations, system call specialization can remove the code paths associated with the never-invoked actions from the system call handler.

4.3 Command-line Specialization

The Linux kernel is configurable at boot time through the so-called *kernel command line*. This command line, which is a string passed to the kernel by the boot loader, consists of a number of (parameter, value) pairs. These parameters, which correspond to global variables in the kernel, can sometimes be set at run time as well. In many cases the user has no control over the boot process of an embedded system, and there is often no desire to ever change the values of these parameters at run time.

Figure 4.3 shows the AWPCFG fragment corresponding to the kernel's implementation of this feature. The `parse_commandline` procedure splits the command-line string in (parameter, value) pairs and passes them to the `process_arg` procedure. In this procedure, the `param_handlers` table is scanned, and if a match for a parameter name is found, the appropriate handler is called with the parameter value as an argument. The handler (for example `set_debuglevel`) then sets the corresponding kernel variable, to be used later on during the execution of the kernel, for example in `some_function`.

There are two main specialization opportunities associated with this boot-time parameter feature. First, if the kernel command line that will be used on the device is known in advance and cannot be changed during the lifetime of the system, we can eliminate all unused parameter handlers from the `param_handlers` table. The developer specifies the desired kernel command line to our link-time binary rewriter, which parses this command line and marks all entries in the `param_handlers` table that are needed for successful parsing of the command line. All other entries from the table can be removed. As a result, the AWPCFG data reachability edges to the superfluous parameter handlers disappear, together with their corresponding unknown control flow edges. The handlers are thus unreachable from the entry point of the AWPCFG, and can be removed by unreachable code elimination.

The second specialization opportunity involves specializing the kernel code for specific values of the configuration variables. If the

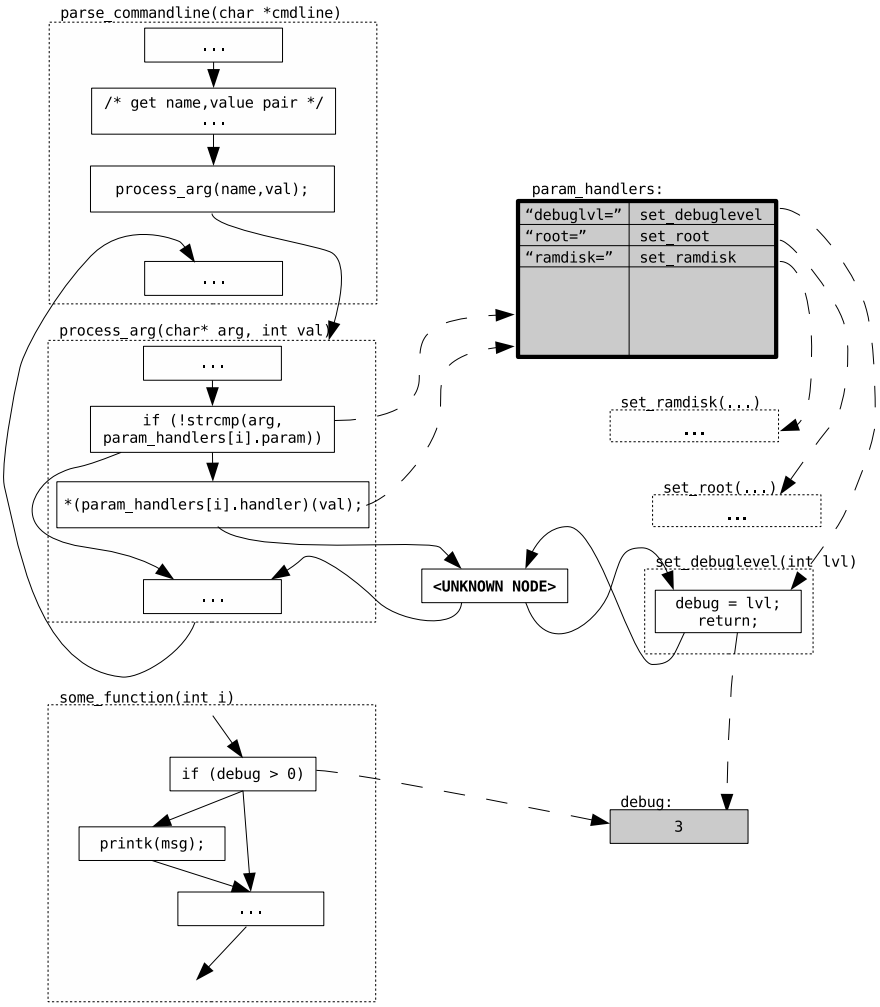


Figure 4.3: Kernel command line handling.

value of a variable is known to be constant throughout the lifetime of the system, the link-time optimizations in our binary rewriter needn't treat this value as unspecified. Instead, the kernel can be optimized for this known value.

As with the elimination of system calls, the developer has to provide a list with additional specialization information. In this case, this list identifies the kernel variables associated with the boot-time parameters and their fixed values. Using symbol information, the link-time kernel specializer looks up the memory locations of the variables, writes the specified values at those locations and marks them as read-only. When constant propagation is then later applied during the generic compaction of the kernel, the desired initial values are propagated into the code and, whenever possible, the code is specialized for those values. In the case of boolean variables, this typically results in compare instructions and conditional branches being removed or replaced by direct branches, thus eliminating unrealizable execution paths.

Suppose that in the example of Figure 4.3 the developer specified that the boot-time parameter `debuglevel` will not appear on the command line, and that `debug`, its associated kernel variable, has a fixed value of 0. Because `debuglevel` will not appear on the command line, its entry can be removed from the parameter handler table. Consequently, `set_debuglevel`, the parameter handler function, becomes unreachable and disappears from the kernel. The `debug` variable now has only one incoming data reachability edge, from a read operation in `some_function`. The developer has also specified that the value of `debug` should be 0, not 3, so 0 is written into this memory location, and because there are no more write accesses to the variable, it can be marked read-only. Afterwards, constant propagation can propagate the known value to specialize the code in `some_function`. The `if`-test and the `printk` call can then be removed from the kernel by the constant propagation optimizations. This example illustrates how small specialization transformations (in this case removing an entry from a table) interact with the established link-time analyses and optimizations to achieve the desired effect.

4.4 Finding Extra Initialization Code and Data

As described in Section 3.3, the Linux kernel is divided into initialization and non-initialization parts. The initialization code and data are removed from memory once the system is initialized and they are no longer needed. The initialization code and data are identified by manually inserted annotations in the kernel source code. This approach has several shortcomings:

- It does not allow for conditional markings: some code is only reachable during system initialization in certain kernel configurations, whereas in others it can also be reached after initialization. This is for example the case for a number of utility functions that are available in the kernel: depending on which features are compiled into the kernel, they are called only during system initialization or also during the steady-state operation of the system. Another example is device initialization code: if the kernel supports hot-plugging of certain device types, the device initialization code can be called at any time. If hot-plugging is disabled, the initialization can only happen during the boot process, and the code will not be needed afterwards. Because the current annotation system does not allow the annotations to depend on the kernel configuration, such code has to be marked "non-init" in all cases.

- There are certain types of data that simply cannot be annotated, because they are defined behind the scenes by the compiler, and never declared explicitly. Examples are literal strings that appear in the code (for example as the format argument for a `printf`-like procedure), and jump tables that are generated by the compiler as part of the translation of a `switch` statement. These kinds of data will always be considered non-initialization data, even though they belong uniquely to initialization code.

- Sometimes initialization code or data is not annotated due to human error: the developer simply forgot to do so, or was not sure whether the code or data are still needed after initialization.

In this section, we present two analyses that identify additional initialization code and data, starting from the code and data that were already identified by the kernel developers.

Initialization Code Detection

The initialization sections are released from memory in a procedure called `free_initmem`. Consequently, all code that is reachable from this procedure, or from its return site, should be considered non-initialization code. This code can be identified with a simple reachability analysis. This analysis should not take into account code paths that go through code that is already marked as initialization code by the kernel developers, as these code paths will cease to exist once the init sections are removed from memory. All code that is not marked in the reachability analysis can be considered initialization code.

Initialization Data Detection

As explained in Section 2.3, data can only be manipulated at the granularity of subsections. Therefore, we will try to move whole subsections at once to the initialization data sections. In general, a data subsection can be considered initialization data if all accesses (i.e., loads and stores) to the subsection occur before the initialization sections are removed from memory. The temporal aspect of this observation is hard to translate to an analysis on the AWPCFG, which provides little temporal information.

A sufficient condition for a data subsection to be considered part of the initialization data is the following: *all accesses to the data should occur either in initialization code or in non-initialization code that was called from initialization code*. This implies that there should be no direct references to the subsection from any non-initialization code or data: if there is to be an access to the subsection from non-initialization code, the address should be passed to this code down the call chain from somewhere in the initialization code. Furthermore, such addresses must not escape, i.e., be stored somewhere in memory so they can be retrieved and used after the initialization sections are released.

For addresses stored in data sections, determining whether they escape involves first identifying where they can be loaded, and subsequently tracing all uses of the loaded values. Due to the complexity and imprecision of alias analysis at link time [Hind01], it is hard to identify with any degree of precision the places where a certain address can be loaded. Therefore, we have opted to limit our analysis to data subsections that are only referenced directly from initialization code. This still includes most of the compiler-generated data that cannot be annotated,

such as jump tables and constant strings.

For our ARM test kernel, only 2396 bytes of the remaining, unanalyzed data are not trivially shown to be non-initialization data. For the i386 test kernel, this is the case for 5447 bytes of data. Note that this does not mean all of this data is initialization data: this is the amount of data for which we would have to implement a complex analysis to determine whether or not it is initialization data. Given the meager potential gains, we feel that this was not worth the effort.

In short, for each data subsection that is not already marked as initialization data, and only has incoming data reachability edges from initialization code, our analysis tries to determine whether the produced address escapes. It does so by tracing all uses of the register into which the address was produced, and of all registers into which derived values are produced (i.e., by adding an offset to the address) throughout the procedure in which the reference occurred. If the address is stored in memory, or passed as an argument to another procedure, it is considered to be escaping.

In fact, assuming an address has escaped whenever it is passed as an argument to another procedure is overly conservative. It is, however, difficult to analyze whether an address escapes over function call boundaries, as in many cases such addresses are passed on to procedures several levels deeper into the call chain (e.g., `printk`, the `printf`-like utility function that is used for kernel output, passes its arguments on to `vsnprintf`, where the actual processing occurs). In the process of passing on the arguments, they are frequently spilled to memory, which means that they are considered to be escaping anyway. This can only be avoided with the use of a good stack analysis that allows to track the stored addresses through the stack as well. We have opted to use a different approach to reduce the imprecision of our analysis: whenever an address is passed on to another procedure, our link-time rewriter checks whether this procedure is in a list of known-safe procedures, i.e., procedures whose arguments are known not to escape. If this is the case, our analysis assumes the address does not escape. Upon termination of the analysis, our rewriter emits a list of (procedure, argument number) pairs for which the analysis assumed the argument escapes. This list can then be used by the developer to identify other known-safe procedures, increasing the precision of the analysis the next time it is run.

Manually verifying whether procedure arguments can escape may

seem to be an onerous task. However, the number of procedure arguments that need to be checked is relatively small: for our ARM test system, only 50 arguments needed to be checked, of which 33 were safe. Using a good source code browser (e.g., the Vim editor combined with the ctags source code indexing utility), a person with reasonable C-language skills but no intimate knowledge of the Linux kernel internals can verify the safety of a procedure argument in less than one minute. Consequently, building the known-safe procedure list for the whole kernel takes less than one hour, and this list can easily be reused even if the kernel is later reconfigured. Unfortunately, the list of known-safe procedures cannot be carried over directly to a new kernel version. The implementation of some procedures may have changed in the new version, and procedures that were previously safe may now have become unsafe.

While moving code and data to the initialization sections does not reduce the kernel's static ROM footprint, it does reduce the static RAM footprint during steady-state operation of the system, and as such it is a desirable optimization.

4.5 Boot Process Specialization

Until the `init` thread is spawned, and the initialization sections are removed from memory, the kernel executes as a single-threaded program. The kernel's execution up to this point is completely determined by the boot-time command line parameters and the hardware that is present in the system. As these two things typically don't change in the course of a device's lifetime, we can conclude that the kernel will execute exactly the same code on each boot. By observing the boot sequence, through code coverage or profiling analysis, we can determine which of the initialization code is never executed, and remove this code from the kernel image. The unreachable code and data detection algorithm in our link-time rewriter will then automatically remove the associated data sections as well. While this optimization will have little or no impact on the steady-state RAM footprint of the kernel (as most of the removed code and data are from the initialization sections), it does reduce the kernel's static ROM footprint. This optimization was first proposed by He et al. [He07].

It is important to note that this optimization is only partially safe. While normally the boot process is exactly the same each time the sys-

tem is turned on, this assumption is no longer true when some hardware peripheral becomes defective. Any error handling mechanisms that were present in the initialization code to cater for such an eventuality will no longer exist after this specialization, and the system will no longer be able to gracefully handle this situation. In some cases, this is not a problem: once part of the hardware breaks down, the system is useless and so there is no need for any part of it to work correctly any more. In other cases however, a more graceful error handling strategy is required and blindly removing all unexecuted initialization code is therefore not allowed. Still, even in this situation part of the unexecuted initialization code — for example code paths in driver code that are only relevant for certain revisions of the hardware — can be removed, under developer supervision. The link-time rewriter identifies unexecuted code regions (e.g., whole procedures or individual code paths within a single procedure) and presents them to the developer, who can then decide on a case-by-case basis whether or not to they may be removed.

4.6 Evaluation

In this section, we evaluate the impact of the proposed specialization techniques on the kernel's memory footprint. Performance measurements performed with LMbench have shown no discernible difference between kernel performance with and without specializations applied, so we will not discuss slowdowns any further in this evaluation section.

Table 4.1 presents the impact of the specialization techniques on the kernel's memory footprint. Each row shows the impact of the specialization mentioned in the left-most column when applied in addition to all transformations in the rows above. Note that the standard link-time optimizations are applied to the kernel as well.

4.6.1 Initialization Code Motion

Identifying extra initialization code has no impact on the kernel's ROM footprint, as the total amount of code in the kernel is not reduced. It does, however, reduce the static RAM footprint after initialization with 1.4% for the i386 and 1.7% for the ARM platform. This corresponds to approximately 5100 and 5800 instructions respectively that were moved to the `.text.init` section.

	text size	data size	init text size	init data size	image size	compressed image size	init mem. footprint	memory footprint
original kernel	714	292	46	8	932	470	1061	1006
+whole-program optimization	632 *-11.6%*	263 *-9.9%*	45 *-2.4%*	8 *-2.2%*	815 *-12.5%*	456 *-2.9%*	948 *-10.6%*	895 *-11.1%*
Specialization techniques								
+initialization code motion	617 *-13.6%*	263 *-9.9%*	60 *29.3%*	8 *-2.2%*	815 *-12.5%*	456 *-3.0%*	948 *-10.6%*	880 *-12.5%*
+initialization data motion	617 *-13.6%*	249 *-14.6%*	60 *29.3%*	22 *169.1%*	815 *-12.5%*	456 *-2.9%*	948 *-10.6%*	866 *-13.9%*
+unused system call elim.	566 *-20.7%*	248 *-15.2%*	60 *29.4%*	22 *169.5%*	763 *-18.1%*	427 *-9.1%*	896 *-15.6%*	814 *-19.1%*
+system call specialization	566 *-20.7%*	248 *-15.2%*	60 *29.4%*	22 *169.5%*	763 *-18.1%*	427 *-9.1%*	895 *-15.6%*	814 *-19.1%*
+command line specialization	566 *-20.7%*	248 *-15.2%*	60 *29.4%*	22 *166.4%*	763 *-18.1%*	427 *-9.2%*	895 *-15.6%*	814 *-19.1%*
+known-constant propagation	565 *-20.9%*	246 *-15.7%*	59 *27.7%*	21 *164.3%*	759 *-18.5%*	425 *-9.5%*	892 *-15.9%*	811 *-19.4%*
+unexecuted init code elim.	563 *-21.2%*	244 *-16.5%*	31 *-32.2%*	9 *14.5%*	715 *-23.3%*	401 *-14.7%*	847 *-20.1%*	807 *-19.8%*

(a) i386

original kernel	1060	203	47	4	1207	575	1315	1264
+whole-program optimization	935 *-11.8%*	203 *-0.2%*	44 *-7.2%*	4 *0.0%*	1041 *-13.7%*	574 *-0.1%*	1185 *-9.8%*	1138 *-10.0%*
Specialization techniques								
+initialization code motion	912 *-13.9%*	203 *-0.2%*	67 *40.7%*	4 *0.0%*	1041 *-13.7%*	574 *-0.1%*	1186 *-9.8%*	1116 *-11.7%*
+initialization data motion	900 *-15.1%*	202 *-0.5%*	77 *63.6%*	4 *17.3%*	1040 *-13.8%*	575 *0.1%*	1184 *-9.9%*	1102 *-12.8%*
+unused system call elim.	847 *-20.1%*	202 *-0.5%*	77 *63.3%*	4 *17.3%*	988 *-18.1%*	546 *-4.9%*	1131 *-14.0%*	1049 *-17.0%*
+system call specialization	846 *-20.2%*	202 *-0.5%*	77 *63.3%*	4 *17.3%*	984 *-18.4%*	546 *-4.9%*	1130 *-14.0%*	1048 *-17.0%*
+command line specialization	846 *-20.2%*	202 *-0.5%*	77 *63.3%*	4 *10.8%*	984 *-18.4%*	546 *-5.0%*	1130 *-14.1%*	1048 *-17.0%*
+known-constant propagation	841 *-20.7%*	202 *-0.5%*	60 *25.9%*	4 *10.8%*	960 *-20.4%*	532 *-7.4%*	1107 *-15.8%*	1043 *-17.4%*
+unexecuted init code elim.	839 *-20.8%*	202 *-0.6%*	39 *-17.4%*	4 *-2.7%*	940 *-22.1%*	520 *-9.5%*	1084 *-17.5%*	1042 *-17.6%*

(b) ARM

Table 4.1: Impact of the specializations on the kernel's memory footprint.

4.6.2 Initialization Data Motion

The identification of extra initialization data has no effect on the ROM footprint either. In this case, the static RAM footprint after initialization is reduced with 1.4% (i386) and 1.1% (ARM), which corresponds to about 14 KiB in both cases. Interestingly, for the ARM kernel, the reduction is mostly achieved by moving bytes from the non-initialization code to the initialization code. As mentioned before, in the ARM kernel the read-only data is subsumed into the code sections, and most of the extra initialization data consists in effect of compiler-generated constant strings and jump tables that could not be annotated as initialization data by the kernel developers.

To achieve this result, we had to manually check a number of procedure arguments in the kernel source code to see whether they escape (see Section 4.4). For the i386 kernel, this required inspection of 30 procedure arguments, of which 11 were found to be escaping. For the ARM kernel, we checked 50 arguments, of which 17 escaped. In both cases, the manual source code inspection took less than one hour.

4.6.3 System Call Elimination and Specialization

To determine which system call handlers may be removed from the kernel, all software that will run on the system has to be analyzed. Thanks to the simple setup of our test systems, this means analyzing just one user-space binary, namely Busybox, and the Linux kernel itself. This analysis was performed using a modified version of our link-time binary rewriter that reports the known register values for each system call instruction during context-sensitive constant propagation. With this approach, we were able to determine for each system call instruction which system call was actually invoked, and we were able to determine all possible values for 14 arguments of 11 system calls for the i386 kernel, and for 15 arguments of 12 system calls for the ARM kernel.

The analysis showed that of the 245 system calls offered by the Linux kernel, only 80 can actually be called in our test configuration. All other system call handlers were removed from the kernel, resulting in a reduction of the static RAM footprint after initialization of 5.2% (i386) and 4.2% (ARM). Note that this is the first optimization that has a discernible impact on the compressed image size, and thus the static ROM footprint in most systems: the i386 kernel's compressed image size is reduced by an additional 6.2% (total reduction 9.1%) , the ARM

kernel's compressed image size is for the first time significantly smaller than that of the original kernel, with a 4.9% reduction.

The impact of propagating the known system call arguments into the kernel is insignificant (0.1% of the non-initialization code was removed for the ARM kernel, and even less for the i386 kernel), but it did result in the removal of a number of argument validity checks in the system call handlers. We were also able to determine all possible values for the command argument of `sys_socketcall`, a multiplexed system call. Of the 17 possible values for the command argument, only 12 actually appeared in the user space programs, so the code paths associated with the other commands could be removed as well.

4.6.4 Command-line Specialization

The next specialization step consists of specializing the kernel for known, fixed command-line parameters. As a first step, we determined which parameters should still be adjustable at boot time. For the i386 system, this was the "root" parameter that specifies the disk device on which the root partition is installed. For the ARM, we additionally left the parameter "console" adjustable, in order to allow us to specify the console input and output devices, as the CerfCube has no screen and keyboard. The auxiliary parsing procedures for all other parameters were removed from the kernel, leading to an additional gain in the initialization data sections of 3.1% for the i386 and 6.5% for the ARM. The overall impact of this first step on the memory footprint is, however, negligible.

The second step was to propagate the known values of the boot-time parameters, and of some driver parameters that can only be changed if the driver is compiled as a module, into the kernel. For this, we selected 29 parameters for the ARM system, and 28 for the i386 system. For the i386 kernel, the impact of this specialization is limited, with a reduction of about 0.3% for all footprint sizes. For the ARM kernel, the results are more pronounced: while the static RAM footprint after initialization only decreases by 0.4%, the uncompressed image size is reduced by 2%, and the compressed image size does even better with a 2.4% size reduction. The better results obtained for the ARM kernel should come as no surprise: the conditional constant propagation analysis, which is the driving force behind the compaction gains attained here, works better for the ARM architecture, where more reg-

isters are available and the run-time stack is much less frequently used for storing results of computations and passing arguments to other procedures.

4.6.5 Boot Process Specialization

The final specialization step involves removing all unexecuted initialization code from the kernel. For this, we need code coverage information that identifies the unexecuted initialization code. This coverage information can easily be derived from the profile data we already collected to limit the performance impact caused by duplicate basic block elimination.

On the i386, 47.6% of the initialization code (both the original code and the extra code we have identified) turns out to be superfluous. For the ARM kernel, this is the case for 41.3% of the initialization code. As a result, the initialization code sections are now considerably smaller than in the original kernel, even though they grew substantially due to the extra initialization code and data identification specializations that were applied before. Moreover, eliminating the unexecuted initialization code also reduces the size of the other kernel sections, in particular those containing initialization data: along with the code, a number of data reference edges disappear, thus disconnecting parts of the AWPCFG so that they can be removed by unreachable code and data elimination.

While this specialization has little impact on the static RAM footprint after initialization, it has a significant impact on the kernel's ROM footprint: the uncompressed image size is reduced with 4.8% for the i386 kernel, and with 1.7% for the ARM kernel. The compressed image size is reduced with 5.2% and 2.1% respectively.

Together, all compaction and specialization techniques reduce the static memory footprint with 20.1% during and 19.8% after initialization for the i386, and with 17.5% during and 17.6% after initialization for the ARM. The uncompressed image size is reduced with 23.3% and 22.1% respectively, whereas the compressed image size is reduced with 14.7% and 9.5% respectively.

Chapter 5

On-demand Code Loading

Code coverage analysis shows that, even after application of the compaction and specialization techniques described in the previous chapters, over 50% of the remaining Linux kernel code is not executed during "normal" system operation. This does not mean all of the unexecuted code can be removed from the kernel. While part of it is indeed useless code that could not be detected due to the limitations of static analysis, there is also a substantial amount of code in the kernel that serves to handle unexpected situations like hardware failures.

In order to reduce the memory overhead of all this unexecuted code, we propose to introduce an on-demand code loading scheme in the kernel. The unexecuted code is removed from the kernel's memory image, but kept available in a repository, and loaded into memory only when it is actually needed. In fact, such a scheme needn't be limited to unexecuted (or *frozen*, terminology introduced by Citron et al. [Citr04]) code: with the right approach, it can be sensibly extended to include all cold (i.e., infrequently executed) code.

In this chapter, we will discuss the issues involved in designing an on-demand code loading scheme for use in an OS kernel. After a short discussion of Linux' loadable kernel modules, which can be thought of as the kernel's standard on-demand code loading technique, we will propose and evaluate two different code loading approaches, and discuss their strengths and weaknesses.

5.1 Design Issues

There are many possible variations on the idea of on-demand code loading, so it is interesting to look at the possible design choices that can be made. First of all, we state a number of design criteria any viable solution should comply to:

- *Reliability*: the correct operation of the kernel must in no way be compromised by the code loading scheme.
- *Performance*: the OS kernel is a performance-critical part of the system, especially as it communicates directly with the hardware devices. Loading code must not slow down the system too much.
- *No changes to source code*: large-scale rewriting or manual annotation of the kernel source code should not be necessary.
- *Automation*: the code to be loaded on demand must be selected and partitioned into loadable fragments automatically. Users of the technique should not need to have an intimate knowledge of the kernel code.

Based on these criteria, a number of design choices have to be made:

- *Code selection*: Will all code be loaded on demand? If not, how do we select which code will always be resident and which code won't?
- *Repository*: From where will the loadable code be retrieved? Possibilities include loading the code over a network, loading it from secondary memory (hard disk, Flash memory) or retrieving it from main memory, where it is stored in compressed form.
- *Trigger*: What mechanism will be used to decide when a code fragment has to be loaded?
- *Buffer management*: Is the buffer into which the code is loaded of fixed or variable size? Is it pre-allocated or allocated at run time on an as-needed basis?
- *Eviction strategy*: Will loaded code be evicted again? If there is code eviction, does it take place immediately after execution of the code or only when there is not enough free memory left? How is concurrent execution within the kernel dealt with?

As it turns out, the eviction strategy is a very important factor in the overall reliability of the code loading scheme. Operating system kernels are inherently multithreaded: even on a single processor, kernel code can be interrupted by other kernel code (for example, an interrupt handler or exception handler that interrupts the execution of a system call handler). When code is selected for eviction, it is important to make sure that the code is not being executed by any thread in the kernel. As the code loading scheme we propose is automatically introduced into the kernel at a very low level (at link time), it is hard to insert locking mechanisms that protect code under execution from eviction. At this low level, there is not enough information about the semantics of the code available to safely introduce locks in such a way that they will provably not result in deadlock or livelock situations. Therefore, alternatives for locking have to be found if loaded code is to be evicted.

5.2 Linux Kernel Modules

The Linux kernel already offers its own built-in code loading scheme. The build system allows developers to compile parts of the kernel (e.g., hardware drivers) as loadable modules. These modules are stored on disk or in ROM and can be loaded either manually with the `insmod` command or automatically when their functionality is needed. Module unloading is done manually, through the `rmmod` command, and reference counting is used to make sure the module's code or data are no longer needed before the module is in effect removed from memory. Modularization of the code is done manually: module initialization and finalization routines have to be written, and the build system has to be made aware of the fact that the code can be compiled as a loadable module. This process requires significant knowledge of the kernel's internals.

Loadable kernel modules are mostly intended for the distribution of kernels for generic machines. Drivers for a wide range of peripherals are compiled as modules, and at boot time the kernel then loads only those drivers that are necessary for the specific hardware it is running on. We feel this scheme is less suited to embedded systems, where the hardware is known in advance and the necessary drivers can easily be compiled into the kernel, obviating the need for the module loading infrastructure altogether and thus reducing the kernel's code size. Furthermore, the module granularity is determined by the kernel's con-

figuration system, and this is not fine-grained enough for our purposes. We wish to remove individual, infrequently executed code paths from a driver or kernel subsystem, and not a driver or subsystem as a whole.

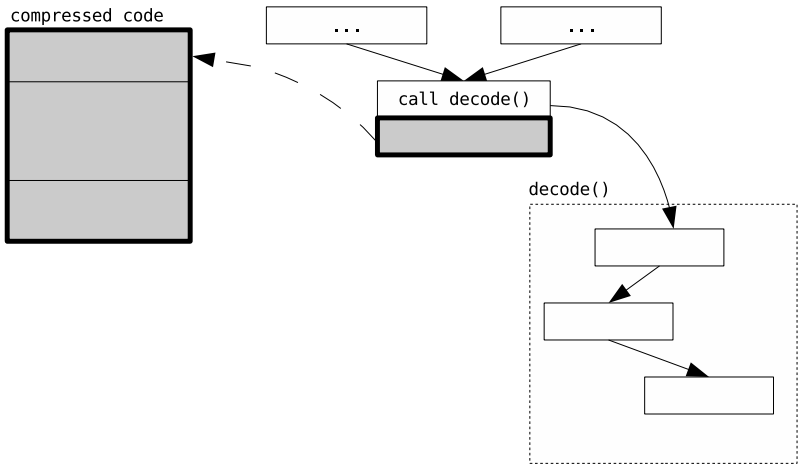
In the next two sections, we propose two alternative code loading schemes that work at a much finer granularity and can be introduced automatically into the kernel: *frozen code compression* and *cold code swapping*.

5.3 Frozen Code Compression

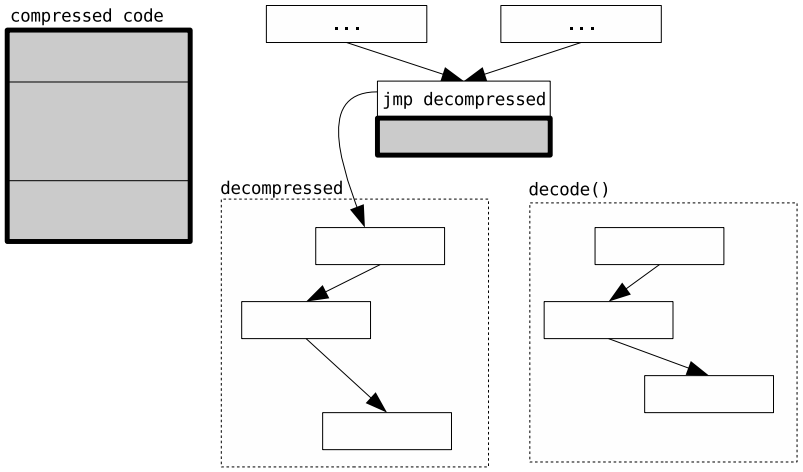
Our first technique is generally applicable and does not require any special hardware support. The code that is to be loaded on demand is partitioned into single-entry regions. These regions are stored in memory in a compressed form, and replaced in the code by a stub that invokes a decompression routine and passes a pointer to the compressed code to it. This situation is depicted in Figure 5.1(a). When the control flow enters the stub, the decompressor is invoked. It allocates a buffer of the appropriate size through `kmalloc`, the kernel's dynamic memory allocation procedure, and decompresses the code into this buffer. The stub is then overwritten with a direct jump to the decompressed code so that subsequent invocations of the code no longer need to pass through the decompressor. Finally, the processor registers are restored to their state upon entry to the stub and control is transferred to the decompressed code. The situation after decompression is shown in Figure 5.1(b).

This scheme side-steps the concurrency issues associated with code eviction by never evicting loaded code from memory. This implies that it only makes sense to compress code that is not expected to be executed, i.e., frozen code. If a code fragment is known to be executed at least once, there is no profit to be made from compressing it, as it will be decompressed and kept in memory anyway.

The downside of this no-eviction strategy is that in the worst case scenario the kernel's memory footprint is still as big as without frozen code compression. This can only be the case, however, if all frozen code is in the kernel for handling exceptional situations (and not because it is undetected unreachable code), and if all possible exceptional situations occur between two consecutive reboots. We consider this to be unrealistic, and hence do not take this situation into account in the remainder of the discussion.



(a) before decompression



(b) after decompression

Figure 5.1: A stub replacing frozen code.

The upside of this approach is that the concurrency issues involved in frozen code compression are significantly simplified, as we will discuss in Section 5.3.5, and that the performance impact of frozen code compression is very small: the decompressor needs to be called at most once for each frozen code region. After decompression, the processor's I-cache and D-cache need to be invalidated to ensure correct execution of the decompressed code. Subsequent invocations of the decompressed code are slowed down only by the extra direct jump they have to perform in the updated stub to reach the decompressed code region. Under normal operation, no code should be decompressed of course, and hence no slowdown will be experienced whatsoever.

The frozen code compression approach is reminiscent of the profile-guided code compression technique proposed by Debray and Evans [Debr02], in which infrequently executed code fragments in user space programs are compressed and decompressed in a pre-allocated buffer whenever they are needed. The scheme proposed by Debray and Evans includes a code eviction strategy, but the authors do not consider the difficulties involved in supporting multi-threaded applications with their approach, and as such do not offer any solution for the concurrency problem we face when trying to apply this technique to an OS kernel.

In the remainder of this section, we will discuss several aspects of this scheme in more detail.

5.3.1 Frozen Code Identification

The frozen code is identified by performing an extensive code coverage analysis on the kernel. The target system is loaded with an instrumented kernel, and subjected to several usage scenarios that the system developer considers "normal" operation. Of course, this may also include some "abnormal" scenarios that, although not normal, occur frequently enough for the developer to consider them important. An example might be the disconnection of devices while they are being used.

As the decompressor calls the `kmalloc` and `kfree` procedures provided by the kernel, we need to make sure that these procedures are never considered frozen to avoid infinite loops. Fortunately, these procedures are so widely used throughout the kernel code that there is no risk of them being considered frozen code in the first place. We have

also decided not to consider any code from the initialization code section as frozen code. While it is technically possible to compress the frozen initialization code as well, we consider this to be useless as this code is removed from memory after booting anyway.

Note that we do not distinguish interrupt handling code from "regular" kernel code, so even the interrupt handlers will possibly be compressed. While the decompression process may cause an interrupt handler to respond slowly to the interrupt when it still needs to be decompressed, this will happen only once. Afterwards, the code will already be decompressed and subsequent occurrences of the same interrupt will be handled as quickly as without code compression. If a slow reaction to even one interrupt is unacceptable, the interrupt handler code has to be identified (either through the execution of more coverage scenarios, or otherwise manually), and excluded from the frozen code.

5.3.2 Frozen Code Partitioning

After the frozen code is identified, it is partitioned into regions that form the basic units of compression. To minimize the amount of bookkeeping information related to the compressed code, we have opted to operate on single-entry regions. As such, one stub and one compressed code address suffice as bookkeeping code and data about each compressed region. If we would have allowed multiple-entry regions, we would need to keep track of the offsets of entry points in the regions as well. Furthermore, when a frozen code region is decompressed, the decompressor has to overwrite all stubs leading to the region with direct jumps to the decompressed code. With multiple entry points, we would have needed multiple stubs per region, and we would have had to keep a list of all stubs associated with each region, so they could all be overwritten at once after decompression. Finally, the compression algorithm we have implemented (which will be described in Section 5.3.4) compresses individual instructions, so the fact that we use single-entry regions, and not larger, multiple-entry regions, has no detrimental effect on the achieved compression ratio.

The partitioning happens in three steps. First, frozen code is partitioned into single-entry regions per procedure. Each frozen basic block that has incoming AWPCFG edges coming from non-frozen code is considered to be the start of a new region. Furthermore, all frozen code

blocks that have incoming AWPCFG data reachability edges are considered to be the start of a new region as well. The latter allows us to relocate all data reachability edges that point to a compressed region to point to the corresponding stub instead. As the address of this stub is known at link time, this can be done without requiring any additional bookkeeping code or data at run time.

In the second step, frozen code regions are merged as much as possible, without violating the single-entry requirement. This implies that any region that has incoming edges from one other region only, is merged with that region. As a consequence, frozen code regions can span procedure boundaries. The purpose of this step is to minimize the number of regions (and, more importantly, stubs) and to minimize the (static) number of inter-region control flow transfers. As the final address of the uncompressed frozen code is not known at link time, inter-region control flow transfers have to be encoded under the worst-case assumption that these transfers can span large displacements in the instruction memory, which is less efficient. The displacements of intra-region control flow transfers are known at link time, by contrast, and as these are often rather small, they can be encoded more efficiently.

A side effect of the region merging step is that, once run-time decompression is triggered, potentially more code than strictly necessary will be decompressed. We consider this to be an acceptable drawback, however, as our main objective is to reduce the kernel memory footprint during normal operation, while still having a reduced, albeit not maximally reduced, size under abnormal operation.

We should note that this partitioning system is much simpler than the one proposed by Debray and Evans [Debr02] for cold code compression. Because their system operates with a small code buffer that stores decompressed code, they need to perform the partitioning of cold code in such a way that it balances the need for a small decompression buffer with the need to minimize the dynamic number of control flow transfers between separate regions. If there are too many control flow transfers between regions, they risk having to decompress previously decompressed, but evicted, code over and over again.

Finally, in a third step we select the partitioned and merged frozen regions that will actually be compressed. In order to achieve a good overall compression ratio, it is important to compress only those regions for which compression actually yields a size reduction. If a

region is too small, or not very compressible, the possibility exists that the combined size of the stub and the compressed region is bigger than the original, uncompressed region. Determining in advance which regions can be compressed profitably is an undecidable problem [Debr02]. Therefore, we resort to a heuristic approach to select the actual regions that will be compressed:

- All regions smaller than a certain minimum size will be excluded from compression. We determine this minimum size by assuming a fixed compression factor γ for all code. Then it is profitable to compress a region if $K + \gamma \times S < S$, with K the stub size and S the uncompressed size of the region. The minimum size S_0 for which this equation holds is $S_0 = K/(1 - \gamma)$.
- After the small regions are discarded, all regions are compressed. Any region for which compression turns out to be unprofitable is then deselected, and will appear in the final kernel as uncompressed code.

Once the code is partitioned into regions, these regions are merged. The profitable ones are selected, and they are patched to remove all inter-region fall-through control flow. This last step removes any memory layout dependencies between selected regions, and thus simplifies the run-time decompression process.

Note that extra precautions have to be taken with regards to the exception handling constructs within the Linux kernel as described in Section 3.7. Whenever an exception occurs, the address of the faulting instruction is retrieved from the exception handler table *through binary search*. This implies that (1) the corresponding data reference edges in the AWPCFG should *always* point to the actual instruction that raised the exception, and should not be moved to a compression stub and (2) because of the binary search the entries in the exception handler table should always be in sorted order. These two implications make it impossible to compress any instruction referenced from the first column of the exception handler table. If the address of the instruction isn't known at link time, it is impossible to sort the exception handler table correctly. Moreover, moving the data reference edge to the stub, whose location is known at link time, is explicitly disallowed in this case. For these reasons, basic blocks with incoming edges from the first column of the exception handler table are explicitly excluded from the partitioning of frozen and executed code. The fixup code, which is ref-

erenced from the second column of the exception handler table, can however be compressed without problems.

5.3.3 Decompression Stubs

The stubs that replace the frozen code regions invoke the decompressor with the address of the relevant compressed code. In order to keep these stubs as compact as possible, the call to the decompressor is performed without arguments, and the compressed code pointer is appended directly to the call instruction (the gray block in Figure 5.1(a)). As such, the pointer is located at the return address of the call to the decompressor, which can load this pointer by simply dereferencing its return address.

To ensure the correct working of the kernel, the values of the processor registers have to be preserved over decompression. To ensure this, the stub has to save all registers it changes on the stack first. For an architecture like the i386 this is no problem, as the procedure call stores its return address on the stack, and no registers are overwritten. On architectures like the ARM that store the procedure return address in a register, this so-called link register has to be stored on the stack first. As such, a stub on the i386 architecture would be 9 bytes large (5 bytes for the call instruction and 4 bytes for the compressed code pointer), whereas a stub on the ARM architecture will be 12 bytes large: 4 bytes for saving the link register, 4 bytes for the call and 4 bytes for the compressed code pointer. Note that there is no need to include an instruction that restores the link register in every single stub, since this can instead be done in the decompressor itself, of which only one instance is present in the final program.

5.3.4 Code Compression and Decompression

In this dissertation, we do not focus on finding the best software-controlled code compression technique, but rather on using compression in the context of an OS kernel. As such, we have decided to use an established compression scheme rather than inventing our own new one. The code compression algorithm we apply is basically the same as the algorithm described in [Debr02], but we apply it on the i386 and ARM architectures instead of on the Alpha architecture. The data that has to be compressed is a sequence of machine code instructions. Each of these instructions consists of an opcode field, followed by a number

```
i = j = v = b = 0
do
     v = 2*v + NextBit()
     b = 2*(b + N[i])
     j = j + N[i]
     i = i + 1
while v >= (b + N[i])
return D[j + v - b]
```

Figure 5.2: Decoding algorithm for a canonical Huffman code.

of instruction-specific fields (the presence of these fields depends on the opcode). For each of the field types, a stream is created containing the field values for each instruction in the sequence. One additional stream is created that consists of the sizes of all frozen code regions. For each of these streams, an optimal Huffman code is generated.

Each individual frozen region is then compressed by first writing the Huffman coded version of its size, followed by the instruction sequence that makes up the region, whereby the value for every instruction field is replaced by its Huffman code.

Decompression is fairly easy. The decompressor first reads the size symbol from the beginning of the stream and decodes it. Then a buffer of the correct size is allocated, and the instructions can be decoded. This is done by repeatedly reading an opcode field from the stream and decoding it. Based on the opcode, the decompressor knows which field types will follow, so it can read the corresponding symbols from the stream one by one. This process is repeated until the whole allocated decompression space is filled.

By using so-called *canonical Huffman codes* [Witt94] it is possible to build a very simple and small decompressor. The canonical Huffman code is one possible variant of the possible Huffman encodings for a given set of tokens. This variant has the property that the code words of length i are the i-bit numbers with the values $b_i, b_i+1, \ldots, b_i+N[i]-1$, where $N[i]$ is the number of code words of length i in the code, $b_1 = 0$ and $b_i = 2 \times (b_{i-1} + N[i-1])$. If we store the tokens to be encoded in an array D ordered by their code word value, the algorithm in Figure 5.2 is a simple and fast way to decode the code words[1].

After the code is decompressed, all inter-region PC-relative control

[1]This is the same representation of the algorithm as was presented by Debray and Evans [Debr02].

flow offsets have to be recomputed. As the final address of a frozen code region is only known at decompression time, it is impossible to compute the inter-region jump and call offsets in advance. An efficient way to perform this recomputation is to insert sentinel instructions (with an invalid opcode) into the instruction stream to indicate which instructions need to have their offsets recomputed. At link time, these offsets are computed as if the frozen code region was located at address 0. During decompression the correct offset can be computed by subtracting the final address of the decompressed code region from the precomputed offset.

5.3.5 Concurrency Issues

The Linux kernel is multithreaded, preemptible and supports symmetric multiprocessing. For all of these reasons, it is possible that different threads have to access the decompressor or decompressed code concurrently. In order to guarantee correct operation of the kernel in all circumstances, some form of locking needs to be implemented. Our major concerns for the implementation of the locking scheme are correctness and performance. Because the decompressor needs to be preemptible to allow other interrupt handlers or higher-priority threads to run during decompression, we need to minimize the locking used to prevent deadlocks and race conditions.

Consequently, our decompressor is implemented in such a way that it is fully reentrant, allowing multiple threads to execute it concurrently. If this weren't the case, the complete decompressor would need to be locked, which would allow priority inversion[2] to occur. This is obviously undesirable.

A decompressor that can be executed by multiple threads at the same time, however, opens the door to possible race conditions. More precisely, two race situations can occur.

First, it is possible for one thread to execute the first instruction of a stub while the decoder in another thread is concurrently overwriting this same stub. This could cause the first thread to execute a partially overwritten instruction. The solution is to overwrite the stub atomi-

[2]Priority inversion occurs when a low priority thread holds a shared resource that is required by a high priority thread. This forces the high priority thread to wait until the low priority thread is finished with the resource, effectively inverting the relative priorities of the two threads.

cally, ensuring that no thread can ever see a partially overwritten instruction. On the ARM platform, atomically overwriting the stub is trivial as we only need to overwrite the first four bytes of the stub, and it is always possible to do a four-byte atomic write. On the i386 platform, the jump instruction we need to write is five bytes long (one byte for the opcode, four bytes for the jump offset), so one atomic write seems insufficient. Adding locking to each stub is unacceptable as well, as this would at least double the size of the stubs. Fortunately, other solutions exist: on a single-processor system, it suffices to disable the processor interrupts while the stub is being overwritten, so that no other thread can interrupt the overwriting code. But on multiprocessor systems, this is not enough, as threads on other processors might still see an inconsistent stub state. A workaround exists, at the cost of increasing the size of each stub with five bytes: we just add a jump to the next instruction at the beginning of each stub. Because the displacement of this jump is 4 bytes long, this displacement can be overwritten in one atomic operation. When the stub is first executed, the jump acts as a no-op. As soon as the displacement is overwritten, the jump instruction functions as the jump into the decompressed code.

The second possible race condition occurs when two threads concurrently enter the decoder from the same stub. This means the same compressed region will be decompressed twice, and the stub will be overwritten twice. While this is a wasteful situation in which code is unnecessarily decompressed multiple times, it does not cause incorrect behavior. We can avoid this situation by creating one lock per region. The decoder then acquires this lock before decompressing the region, and if a second instance of the decoder is invoked for the same region, it only has to wait until the lock is released and then jump back to the (now overwritten) stub. While this is the best solution for performance, it requires keeping a lock for each region, which wastes a considerable amount of space only to guard us from possibly (but improbably) wasting space at some point in the future. Therefore we have opted for a different solution that is somewhat slower, but does not require a lock for each region. The region is always decompressed into a fresh buffer but before the stub is overwritten the decoder checks whether some other thread has already done so. If this is the case, the race condition has occurred and the current thread releases its decompression buffer (via the kernel's built-in `kfree` procedure) and jumps to the already-overwritten stub. In order to avoid introducing new race conditions, the checking and overwriting of the stub is protected by a lock.

```
PlaceSections(0)
S = ComputeCompressedSize()
S' = 0
while S > S':
     S' = S
     PlaceSections(S)
     S = ComputeCompressedSize()
PadCompressedSection(S' - S)
```

Figure 5.3: Section placement algorithm in the presence of a compressed code section.

5.3.6 Section Placement

On architectures like the i386 and the ARM that do not have a centralized Global Offset Table [Sriv94], explicit addresses appear in the code nearby or in the instructions that use the addresses. On the i386, they appear as immediate operands in the instruction. On the ARM, they appear in data blocks that are intermingled with the code. Addresses that appear like this in frozen code regions have to be compressed as well. This leads to an interesting phase ordering problem: the final size of the compressed code region depends on its contents, and thus on the addresses that appear in the region. However, these addresses depend in turn on the size of the compressed code, as the placement of the code and data sections following the compressed code section depends on the size of this section.

We solved this problem with an algorithm that iteratively places the sections based on an estimate of the compressed code size. After each placement round, this estimate is recomputed. The algorithm stops when the new estimate is lower than the previous one. At this point, the compressed code will fit in the space that is reserved for it. The compressed code is then padded to fill the complete reserved space in order to ensure that the addresses do not change later on when the binary image of the compacted kernel is generated. The algorithm is shown in Figure 5.3. `PlaceSections(x)` assigns addresses to all sections, assuming size x for the compressed code section. `ComputeCompressedSize()` adjusts the addresses in the frozen regions and computes the compressed code section size. `PadCompressedSection(x)` appends x bytes to the section.

5.4 Frozen Code Compression Evaluation

In this section, we will evaluate the impact of the frozen code compression technique on the kernel's footprint and performance.

For the coverage analysis, we mostly stressed the web server functionality and network interaction of our test systems. They were subjected to a number of usage scenarios: long idle time, high load web serving, low load web serving, requests for non-existing web pages, etc. Failure conditions like network errors and file system corruption were considered to be abnormal behavior, and as such they were not included in our usage scenarios. The code for handling these failure conditions was consequently considered frozen. The correctness of the kernel after frozen code compression was tested by triggering some conditions that did not occur during the code coverage analysis, like unplugging the network cable during transfer of a large file, running a file system check on the system's hard drive, etc. Furthermore, we executed a large number of system utility programs that were not executed during the coverage analysis. These tests resulted in considerable amounts of code being decompressed, without incorrect kernel behavior being observed.

In order to estimate the effectiveness of the frozen code identification, we observed our test systems during a five-day period. In this period, the i386 system decompressed 10 regions for a total of 4502 bytes and the ARM system decompressed 4 regions for a total of 2144 bytes. In both cases, the decompressed code was responsible for handling a UDP connection request to the systems. During the coverage analysis there were no UDP connection requests to either of the two systems, and we do not consider them to be normal behavior either because the systems only serve as HTTP servers, and HTTP is a TCP service. Consequently we conclude that the coverage analysis performed on our test systems is sufficiently accurate for this test case.

5.4.1 Impact on Kernel Footprint

Table 5.1 presents the impact of frozen code compression on the kernel's memory footprint. Here, the frozen code compression is applied on top of all previously described compaction and specialization transformations. The next-to-last row of each subtable shows results for a kernel to which unrestricted basic block factoring was applied. The

	text size	data size	init text size	init data size	image size	compressed image size	init mem. footprint	memory footprint
original kernel	714	292	46	8	932	470	1061	1006
+whole-program optimization	632 *-11.6%*	263 *-9.9%*	45 *-2.4%*	8 *-2.2%*	815 *-12.5%*	456 *-2.9%*	948 *-10.6%*	895 *-11.1%*
+specialization	563 *-21.2%*	244 *-16.5%*	31 *-32.2%*	9 *14.5%*	715 *-23.3%*	401 *-14.7%*	847 *-20.1%*	807 *-19.8%*
Compression								
+frozen code compression	276 *-61.4%*	466 *59.5%*	31 *-32.2%*	9 *14.5%*	652 *-30.1%*	447 *-5.0%*	782 *-26.2%*	742 *-26.3%*
Restricted duplicate basic block elimination								
+profile information	276 *-61.3%*	466 *59.5%*	31 *-32.2%*	9 *14.5%*	652 *-30.1%*	447 *-4.9%*	783 *-26.2%*	742 *-26.2%*

(a) i386

	text size	data size	init text size	init data size	image size	compressed image size	init mem. footprint	memory footprint
original kernel	1060	203	47	4	1207	575	1315	1264
+whole-program optimization	935 *-11.8%*	203 *-0.2%*	44 *-7.2%*	4 *0.0%*	1041 *-13.7%*	574 *-0.1%*	1185 *-9.8%*	1138 *-10.0%*
+specialization	839 *-20.8%*	202 *-0.6%*	39 *-17.4%*	4 *-2.7%*	940 *-22.1%*	520 *-9.5%*	1084 *-17.5%*	1042 *-17.6%*
Compression								
+frozen code compression	367 *-65.3%*	533 *161.8%*	40 *-16.3%*	4 *-2.7%*	797 *-34.0%*	557 *-3.0%*	943 *-28.2%*	900 *-28.8%*
Restricted duplicate basic block elimination								
+profile information	368 *-65.3%*	533 *161.9%*	40 *-16.3%*	4 *-2.7%*	801 *-33.7%*	557 *-3.0%*	944 *-28.2%*	901 *-28.7%*

(b) ARM

Table 5.1: Impact of frozen code compression on the kernel's memory footprint.

		i386	ARM
without frozen code compression	*non-init code size (KiB)*	563.12	839.35
	# instructions	183588	214729
partitioning overhead	*code size growth (KiB)*	3.37	11.94
	total non-init code size before compression (KiB)	566.49	851.29
frozen code	*# regions*	1326	1714
	# instructions	99611	129351
	frozen code size (KiB)	303.42	505.28
	compressed size (KiB)	221.79	329.33
	compression ratio	0.73	0.65
decompressor overhead	*total stub size (KiB)*	11.65	20.09
	decompressor code size (KiB)	1.14	1.39
	decompressor data size (KiB)	0.32	0.02
	section placement padding (KiB)	0	1.02
	net gain (KiB)	65.14	141.49

Table 5.2: Effect of frozen code compression on the kernel's code and data size.

last row of each subtable shows the impact of restricting duplicate basic block elimination by means of profile information. This last row is included as the performance degradation caused by both kernel versions (rewritten with and without profile information) is compared in Section 5.4.2.

Table 5.2 shows the impact of the frozen code compression in more detail. After all compaction and customization techniques are applied, the i386 kernel has 563 KiB of non-initialization code and the ARM kernel has 839 KiB. This corresponds to 183588 and 214729 instructions respectively. Splitting the code in frozen and non-frozen parts and partitioning the frozen code in single-entry regions incurs some overhead: a number of jump instructions have to be inserted to ensure correct control flow. Furthermore, because the final addresses of the frozen code regions are unknown, some constructs have to be encoded less efficiently. For instance, on the i386 inter-region jumps will always have a four-byte offset whereas in some cases it would have been possible to use the shorter one-byte-offset jump instruction if the code hadn't been split up. On the ARM platform absolute addresses cannot be generated in one instruction and the efficiency of the instruction sequences that can be used to generate an address is dependent on the code layout [DeSu07]. In the frozen code regions, one always has to use the most conservative (meaning longest) instruction sequence. For the i386 platform, the total partitioning overhead is 3.37 KiB, for the ARM platform it is 11.94 KiB.

For the i386, 54% of the non-initialization code was considered frozen and partitioned into 1326 single-entry regions. For the ARM, 59% of the non-initialization code was frozen and partitioned into 1714

regions. These frozen code percentages are lower than those reported by Cours et al. [Cour04]. This was to be expected, as the preceding link-time compaction and specialization transformations have already removed part of the unreachable code from the kernel.

After compression, the frozen code for the i386 kernel is 221.79 KiB large, which means a compression ratio (= size compressed code/size original code) of 0.73 is achieved. For the ARM kernel, the frozen code size after compression is 329.33 KiB, so the compression ratio is 0.65. While the compression ratio for ARM code is on par with the ratio reported by Debray and Evans [Debr02] for a similar compression scheme for the Alpha architecture, significantly less compression was achieved for the i386 code. This should be no surprise, as i386 CISC code is typically much denser than the RISC code of the Alpha and ARM architectures, leaving less opportunities for compression.

Taking into account the overhead incurred by the stubs, the code and data size of the decompressor and the padding bytes added for section placement (see Section 5.3.6), a net gain of 65.14 KiB is achieved for the i386 kernel and 141.49 KiB for the ARM kernel. As shown in the next-to-last row of Table 5.1(a) and (b), this results in an additional reduction of the static RAM footprint after initialization under normal operation of 6.5% for the i386 kernel and 11.2% for the ARM kernel. Similar size reductions (6.8% and 11.9% respectively) are achieved for the uncompressed kernel image.

Given the compression ratio and stub size for both architectures, we can now determine the minimum region size for compression as explained in Section 5.3.2. According to the formula explained there, the minimum region size for the i386 architecture should be $9/(1-0.73) \approx 34$ bytes, for the ARM architecture this is $12/(1-0.65) \approx 35$ bytes. However, brute force searching shows the best minimum size to be 50 bytes for both architectures, so our measurements were done with this minimum region size. In both cases, the difference in results was, however, smaller than 0.2%, which shows that the formula does give a good approximation for the optimal minimum region size.

Combining all compaction and specialization transformations with frozen code compression allows us to reduce the kernel's static RAM footprint after initialization by 26.3% for the i386 architecture, and by 28.8% for the ARM architecture. The size of the uncompressed image is reduced by 30.1% and 34% respectively. Disappointingly, frozen code compression partially undoes the size reductions on the compressed

	unzipped	gzipped	ratio
i386 all code	576631	336132	0.58
i386 non-frozen code	282484	166607	0.59
i386 compressed frozen code	227112	216730	0.95
ARM all code	859496	483956	0.56
ARM non-frozen code	376312	204973	0.54
ARM compressed frozen code	338272	317470	0.94

Table 5.3: Gzip compression ratios for uncompressed and compressed code.

kernel image: these drop from 14.7% and 9.5% to 5% and 3% respectively.

In order to understand the cause of this loss, we have applied gzip compression separately on the completely uncompressed code sections, on the non-frozen code sections and on the compressed frozen code sections of the kernel for both architectures. The results are shown in Table 5.3. This table clearly shows the cause of the problem: the remaining uncompressed code is just as easily compressible as it was before, but the already-compressed code is almost not further compressible at all. As our instruction compression algorithm is less powerful than gzip's algorithm (compression ratio 0.73 and 0.65 versus 0.58 and 0.54), the net effect is that the total gzipped size of all code grows with 47 KiB for the i386 kernel and 38 KiB for the ARM kernel, which accounts for all of the drop in compressed image size reduction.

Even more than with duplicate basic block elimination, the viability of frozen code compression as a compaction strategy depends on the ultimate optimization goal. While this technique can significantly reduce the kernel's static RAM footprint, it should not be used if the main optimization goal is to reduce the kernel's compressed image size.

5.4.2 Impact on Kernel Performance

Figure 5.4 shows the performance degradation of fully rewritten (i.e., all optimizations and specializations and frozen code elimination enabled) kernels when compared to the original kernels for the two architectures. As we had already shown in Section 3.8.3, unrestricted duplicate basic block elimination has a significant impact on the kernel's performance. Therefore, we have included for each architecture performance measurements for a kernel that was rewritten with unrestricted duplicate basic block elimination, and for one that was rewritten with profile-guided duplicate basic block elimination.

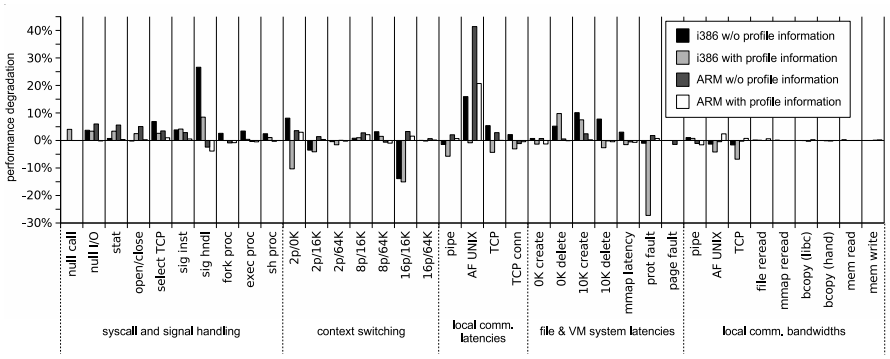


Figure 5.4: Performance degradation for the LMbench benchmark suite.

For the i386 kernel, the average performance degradation was 2.46% with unrestricted basic block factoring, while the kernel with restricted basic block factoring experienced a 1.04% speedup on average. For the ARM kernel, the average performance degradation was 2.07% and 0.67% respectively. In general, these numbers are comparable with those reported in Section 3.8.3, so we can conclude that the frozen code compression technique does not incur a significant slowdown.

For completeness, we have included the kernels with restricted duplicate basic block elimination in the last row of Table 5.1 (a) and (b). Again, we note that restricting duplicate basic block elimination has very little impact on the kernel's footprint, while having a noticeable effect on kernel performance, so whenever possible the binary rewriter should in effect be provided with profile information.

In general, we can conclude that the frozen code compression technique is a viable technique for reducing the kernel's static RAM footprint, and for reducing the size of the uncompressed image. When the optimization goal is the minimization of the compressed image size, this technique is not very useful. The principal downside of this technique is its no-eviction policy, which limits the technique to loading only frozen code, and which makes it impossible to determine a hard upper bound on the kernel's memory usage. For this reason, we have developed a second technique, that does not suffer from these limitations. This technique will be discussed in the next section.

5.5 Cold Code Swapping

Virtual memory paging [Henn03] allows programs to use more memory than is physically present in the system, by swapping out pages to a secondary storage medium, usually a hard disk. When there is a memory access to a page that is not present in physical memory, the processor's memory management unit generates a page fault. The page fault handler, which is part of the operating system, is then responsible for retrieving the page from the swap space and placing it somewhere in memory. Because of the high latency involved in reading a page from disk, the OS usually puts the process causing the page fault to sleep and schedules another process to run in its place while the page is retrieved. This makes swapping less suitable for use in the OS kernel itself, and indeed Linux does not implement swapping for kernel memory. While this has been repeatedly proposed in the past, the kernel developers rejected the idea because of the amount of timing-critical code in the kernel that is not allowed to sleep. Separating this timing-critical code and data from other code and data would be too involved and error-prone to be practical[3].

Our second code loading technique will introduce a limited, directed form of virtual memory paging for the kernel memory. By limiting the swapping to kernel code (which is read-only, obviating the need for slow write-back operations), and using a faster secondary storage medium to store the swapped-out pages, we believe our approach is not susceptible to the drawbacks of kernel memory swapping described above.

Essentially, our scheme separates the code to be loaded on demand from the always-resident code and places it on separate virtual memory pages. These pages are not mapped into physical memory, so when the code is needed, a page fault will occur. A modified page fault handler will distinguish this kind of page fault from "regular" page faults that are caused by user-space programs, and load the necessary code from the secondary storage medium. It is essential that the code repository can be accessed sufficiently fast, so the thread causing the page fault does not have to be put to sleep while the page is loaded.

In order to reduce the execution speed penalty incurred by the code loading, our scheme only swaps out infrequently executed (cold) code.

[3]The relevant thread on the Linux Kernel Mailing List can be viewed at `http://lkml.org/lkml/2001/4/17/115`

To further reduce the number of code loading events, a good code placement strategy is needed that places related code on the same virtual memory page as much as possible.

There is a fixed-size buffer of pre-allocated physical frames available to store the loaded pages. Like regular virtual memory, the buffer is managed with a replacement policy, for example round robin. Code eviction occurs whenever the buffer is full, at which time the replacement policy decides which page will be removed from memory without checking whether the code on the page is being executed at eviction time. Still, this does not raise concurrency issues, as will be illustrated in Section 5.5.1.

There are several possible alternatives for the secondary storage medium. For example, similarly to the approach taken with the frozen code compression technique, the unloaded code could be compressed and stored in memory. In the further discussion, however, we will focus on the case where the unloaded code repository is stored in Flash memory. This is a reasonable choice, as Flash memory is already available in most embedded systems for storing the firmware. The read speed of Flash memory is sufficiently fast to avoid having to put a thread to sleep because a page of code has to be loaded. For currently available Flash memory parts (Intel Embedded Strataflash P33[4]), a 4 KiB page can be read in approximately 40 microseconds. As only code is ever swapped in, there is no need to write back pages to Flash memory when they are evicted from memory, avoiding costly Flash write operations that would slow down the process.

Note that it would also be possible to extend this technique to the kernel's read-only data (i.e., strings for error description). However, we have left this extension for future work. Extending the technique to incorporate writable data is not advisable, as Flash memory wears down after too many write cycles. Consequently, the repeated write-back operations that swapping out writable data would entail, would severely limit the device's lifetime. Furthermore, the write-back operations would significantly slow down the swapping process.

When compared to the frozen code compression technique, cold code swapping has two advantages: there is a hard upper bound on the kernel code memory usage, and the technique can be applied to all cold code, not just the frozen code, potentially resulting in greater memory savings. The main disadvantages are the need for virtual memory sup-

[4] `http://www.intel.com/design/flcomp/datashts/314749.htm`

port, which is not provided by all embedded system chipsets, and the fact that cold code swapping may be slower than frozen code compression: because of the fixed buffer size, it is possible that two or more threads compete for the buffer space, and thrashing occurs. This is never possible with the frozen code compression technique.

In the remainder of this section, we discuss a number of aspects of this technique in more detail.

5.5.1 Concurrency Issues

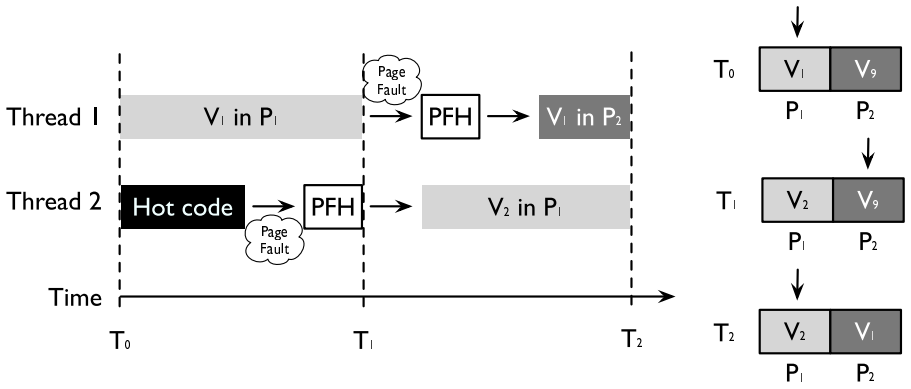


Figure 5.5: An example of the concurrency issues involved in evicting code from memory.

Even though code eviction is done without checking whether the evicted code is being executed, our code loading scheme works reliably. This is illustrated in Figure 5.5. The left side of the figure shows a time line of the execution of two different threads, the right side shows the contents of the buffer at times T_0, T_1 and T_2. The downward-pointing arrow indicates the next page to be replaced according to the replacement policy.

Assume there are two threads in the kernel, the buffer can hold two pages, and we use a round-robin replacement policy. Thread 1 is executing cold code from virtual page V_1 in the physical buffer frame P_1, while thread 2 is executing hot, non-swappable code. At some point, the second thread has to execute some cold code from virtual page V_2, which is currently not in the buffer, causing a page fault. The page fault handler runs in the execution context of thread 2, locates the nec-

essary code in secondary memory and because of the round-robin replacement policy decides to put V_2 in P_1. Once V_1 is unmapped from memory, a page fault will occur in thread 1 when the next instruction in this thread is loaded. The page fault handler will run in the context of thread 1, find V_1 in secondary storage and map it in P_2 because of the round-robin policy, after which execution in thread 1 can continue as before. While this scenario means that thread 1 has been temporarily interrupted, the integrity of the execution has not been compromised. In the worst case, this scenario could cause a cascade of swap-ins for all kernel threads, but it is easily shown that this thrashing will not occur as long as there are at least as many buffer frames as there are kernel threads.

5.5.2 Code Selection

As mentioned previously, only infrequently executed code will be considered for on-demand code loading. Based on basic block profile information gathered for the kernel (the instrumentation technique is discussed in Appendix A), the kernel code is divided into three categories:

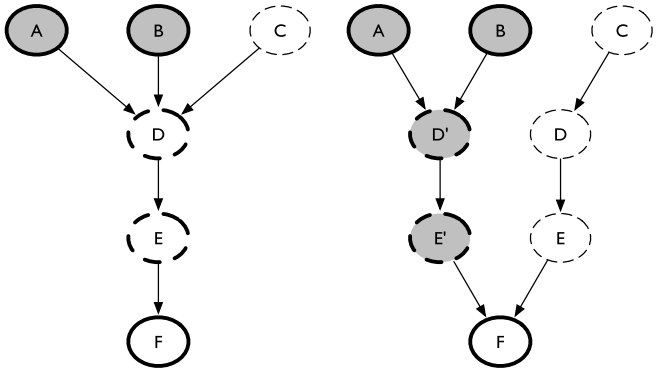
1. The *core* code: this is the code that always has to be present in memory for the system to work correctly. Basically, this portion of the code consists of all code that can be executed before our code loading mechanism is initialized, the page fault handling mechanism and the code needed to read the secondary storage medium.

2. The *base* code: this is the frequently executed (hot) kernel code, which we want to keep permanently resident for performance reasons, even though there are no technical difficulties in swapping it out.

3. The *swappable* code: this is the remaining code, which is either infrequently (cold) or never (frozen) executed. This is the code that will be removed from the kernel image and stored on the secondary storage medium for on-demand loading.

As with the frozen code compression scheme, we will not apply the cold code swapping technique to the initialization code in the kernel. By the time the user space processes start executing, and the device's full memory capacity is needed, the initialization code will already be

removed from memory, making it useless to load this code on demand. Consequently, we will consider all initialization code to be part of the core code.



(a) before duplication (b) after duplication

Figure 5.6: A slice of the call graph before and after procedure duplication. Gray nodes represent initialization code, white nodes non-initialization code. Nodes with heavy borders are considered core code, those with a dashed border are infrequently executed, those with a solid border are frequently executed.

As mentioned before, all code that can be executed before our code loading mechanism is initialized has to be considered core code. While most of this code is part of the initialization code section, it also includes a number of non-initialization utility procedures that are called from the initialization code. We can reduce the amount of non-initialization core code by duplicating all non-initialization procedures that are only called from initialization code prior to the initialization of our code loading mechanism. All calls from initialization code are moved to the duplicated procedures, which can then be considered initialization code as well. The original procedures will then no longer be called prior to the initialization of our mechanism, and can be considered swappable. As the initialization code is released from memory during the boot process, the duplicated procedures incur no memory overhead during the system's steady state operation.

Figure 5.6 illustrates this process. In part (a) we see a slice of the kernel's call graph before duplication. Non-initialization procedure D is called by initialization procedures A and B before the code loading

mechanism is initialized. The call from non-initialization procedure C can only occur after the mechanism is initialized. Because of the calls from A and B, D and its descendants in the call graph E and F must be considered core code. In part (b) the situation after duplication is shown. F is not duplicated as it is hot code, and will not be swapped out anyway. The duplicated procedures D' and E' are only reachable from initialization code, and can thus be considered initialization code themselves. The original procedures D and E can now only be called after the code loading mechanism is initialized and can hence be considered swappable instead of core code.

5.5.3 Code Placement

The problem of code placement to minimize page faults has been studied before. An overview of the existing literature can be found in Section 6.4. All existing algorithms use some variation on run-time profile data as input, and of course they concentrate on achieving a good placement for the most-frequently executed code. As we only have to place the least-frequently executed code, for which there is much less profile information available, these algorithms are not guaranteed to achieve good results. This is especially true in the case where only frozen code is considered swappable, because for this code all execution counts are zero.

Therefore, we have implemented two different code placement algorithms. The first makes use of whatever profile information is available to achieve a good placement, whereas the second aims to minimize, for each entry point in the swappable code, the total number of pages needed to load all swappable code that is directly reachable from that entry point. The second algorithm makes no use of profile information and relies only on an analysis of the static structure of the code. Both algorithms assume that the swappable code can be placed independently from the base and core code, i.e., there are no fall-through control flow paths connecting swappable code to other code. This can easily be achieved in practice by breaking up all fall-through paths in and out of the swappable code by inserting direct jump instructions.

The profile-based algorithm

In this algorithm, the code is placed with a *chain* granularity. A chain is a set of basic blocks that have to be placed in a predetermined or-

der because of control flow dependencies (e.g., fall-through paths or a function call and its corresponding return site). Control flow between chains is always explicit, in the form of function calls, returns or jumps. Consequently, the order of the chains is not important for the correct working of the code.

We use a graph representation of the problem as proposed by Ferrari [Ferr74]. The graph nodes represent chains and are labeled with the size of the chain in bytes. The (undirected) graph edges represent direct control flow between chains. The edge weights are computed by the following formula:

$$weight(e_{ij}) = \sum_{e \in (E_{i \to j} \cup E_{j \to i})} (1 + execcount(e))$$

where $E_{i \to j}$ is the set of direct control flow edges from chain i to chain j and $execcount(e)$ is the traversal count of control flow edge e according to edge profile information. In our current implementation, the edge profiles are estimated from the basic block profile information we have available. It is also possible to obtain exact edge profiles by inserting the appropriate instrumentation into the kernel, just like we did for obtaining the basic block profiles, but, as will be shown in the evaluation section, the estimated edge profiles are accurate enough to derive a good code placement. The traversal counts are incremented by one to ensure that the substantial body of frozen code in the kernel, whose edge traversal counts are zero, is not ignored during placement. If each node is placed on a separate virtual memory page, the graph's total edge weight is an estimate of the number of page faults that will occur at run time.

The nodes in the graph are clustered in such a way that node sizes never exceed the virtual memory page size. This is done in three steps:

1. We try to minimize the total edge weight of the graph. This is done with a greedy heuristic by iteratively selecting the heaviest edge whose head and tail can still be merged without exceeding the page size. In case of a tie, we select the edge with the maximum $commonweight$, which is defined as:

   $$commonweight(e_{ij}) = \sum_{k \in succ(i) \cap succ(j)} (weight(e_{ik}) + weight(e_{jk}))$$

   where $size(i) + size(j) + size(k) \leq$ PAGESIZE. In this way, we try to obtain a graph with less, but heavier edges instead of one with

many light edges. After this step, the total edge weight cannot be reduced any further.

2. We try to maximize the weight of individual edges by iteratively merging sibling nodes (nodes not connected to each other but connected to a common third node). In each iteration we select the nodes for whom the sum of the weights of the edges connecting them to their common parent is maximal. The idea behind this step is that, if more than one page is available in the swap-in buffer, the probability of page j already being in the buffer upon a control transfer from page i is proportional to $weight(e_{ij})$.

3. For each connected subgraph, nodes are merged with a best-fit algorithm. This step minimizes the total number of pages needed for each connected subgraph. We do not yet merge nodes from different subgraphs, because we do not want to pollute the pages for one connected subgraph with code from another subgraph. After all, the likelihood that node j is needed in memory before node i is removed from the swap-in buffer is higher if i and j belong to the same connected subgraph.

The per-entry point minimization algorithm

This algorithm makes no use of profile information to guide the code placement. The swappable code is first partitioned into single-entry regions that do not span procedure boundaries. These single-entry regions (henceforth simply called regions) are the basic units of code placement. Regions that have incoming control flow edges from base or core code are called *entry points*. As a simplification, we assume that the entry points are independent of each other, i.e., that the fact that entry point i was entered at time T has no influence on the probability of any specific entry point j being entered at time $T' > T$. Under this assumption, it makes sense to place the code in such a way that only a minimal amount of pages is reachable from each entry point. After all, in the absence of meaningful profile information we have to assume that all code paths through cold code are equally likely to be followed, so we cannot favor one code path over another for placement on a minimum number of pages.

Initially, each region is placed on its own page. Let P be the set of

pages, and E the set of entry points ($E \subseteq P$). We define two functions:

$$\forall p \in P : entries(p) = \{e \in E \mid p \text{ is reachable from } e\}$$

and

$$\forall e \in E : pcount(e) = \sharp\{p \in P \mid e \in entries(p)\} \ .$$

$entries(p)$ returns the set of entry points from which code on a page p is reachable, *without passing through hot or core code*. $pcount(e)$ computes the number of pages that are reachable from entry point e.

The code placement algorithm tries to minimize the $pcount$ for each entry point by iteratively executing the following steps:

1. Build the set M containing the entry points with maximal $pcount$.
2. Select pages p_i and p_j such that $size(p_i) + size(p_j) \leq$ PAGESIZE and p_i and p_j have a maximum number of entry points in common with M and each other, i.e., $\sharp(M \cap entries(p_i) \cap entries(p_j))$ is maximal. In case there are multiple eligible pairs, select the pair that has the most entry points in common. Stop if no pair can be found.
3. Merge pages p_i and p_j.

Reducing fragmentation

Both described code placement algorithms will terminate with a lot of remaining small pages that aren't merged because there are no direct control flow edges between the code on the pages. In order to reduce fragmentation, a post-pass merges these small pages. We have tried two merging strategies. The first is just best-fit merging of pages, assuming there is no correlation between code on pages that are not connected through direct control flow edges. The second strategy attempts to place the most-frequently executed code together on the same page. By concentrating the hottest code on a small number of pages, we hope to reduce the kernel's working set.

5.5.4 The Modified Page Fault Handler

The original kernel's page fault handler has to be extended to support the code loading mechanism. Our modified page fault handler acts as

a wrapper around the original one: all page faults unrelated to the code loading mechanism are passed on to the original handler.

During system initialization, a number of physical frames are reserved for use as our mechanism's swap-in buffer. Whenever a page fault exception occurs, our handler checks whether the memory access causing the page fault is part of the swappable code's address range. If not, the page fault is passed on to the original handler. Otherwise, the needed page is located in the secondary storage and copied to a physical frame in the swap-in buffer. Then, the page tables are updated and the faulting instruction is re-executed.

For simplicity, the swap-in buffer is not managed by the kernel's built-in page replacement policy. The modified page fault handler implements its own, rather simple, replacement policy. We have implemented three basic replacement algorithms: round robin, random and not recently used (NRU). The latter is implemented by periodically resetting the `accessed` flag of the page-table entries and flushing the Translation Lookaside Buffer (a special-purpose cache that aids the processor in performing virtual address translations). When the buffer is full, the page to be evicted is chosen randomly from those that have an unset `accessed` flag.

5.6 Cold Code Swapping Evaluation

Due to practical limitations, the cold code swapping technique has until now only been evaluated for the i386 architecture. As our i386 test platform does not have Flash memory, we have devised an alternative way to quantify the performance degradation incurred by loading code from Flash memory. In short, the swapped-out code resides in a repository in RAM, and swapping in a page is done by copying it from the repository to the swap-in buffer and mapping it at the correct address in virtual memory. The delay that would be caused by reading the page from Flash instead of from memory is simulated by a delay loop that was inserted in the copying code. In order to determine a realistic delay value, we have performed measurements on our ARM test system, which does have Flash memory on board (Intel StrataFlash J3 NOR-based Flash memory). Reading a contiguous 4096-byte block from Flash takes approximately 442 μs on this device, so this is the delay value we have used for our further experiments.

In order to quantify the effect of the different code placement algo-

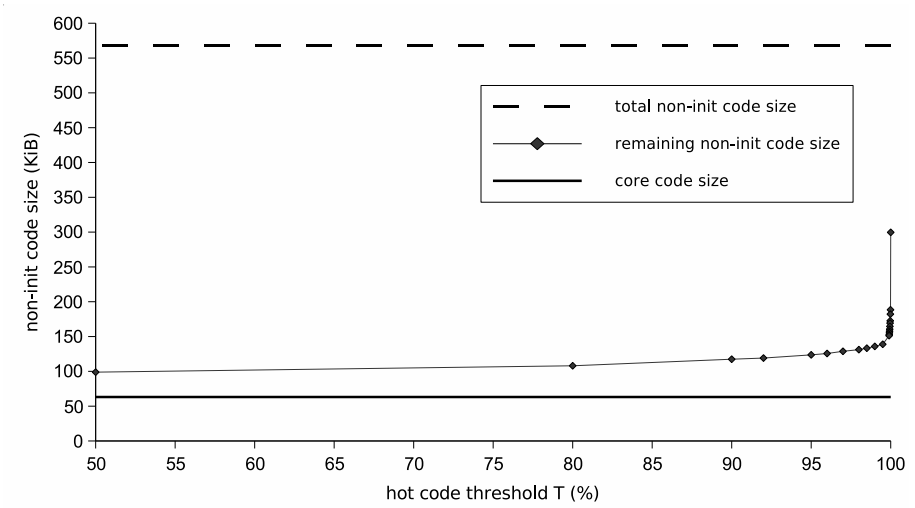


Figure 5.7: Effect of the hot code threshold on the remaining non-initialization code size.

rithms and the other parameters of this code loading scheme (swap-in buffer size and in-kernel page replacement algorithm), we have executed five consecutive runs of the LMbench benchmark suite with a number of rewritten kernels and recorded the amount of page faults (i.e., code loading events) that occurred during execution. The profile information used for guiding the swappable code selection and placement was collected by running the LMbench suite on an instrumented kernel as explained in Appendix A.

5.6.1 Influence of the Hot Code Threshold on the Remaining Code Size

Figure 5.7 shows the effect of the *hot code threshold* T on the remaining size of the non-initialization code in a rewritten kernel. The T value indicates how much of the kernel code is considered hot. For example, $T = 95\%$ means that the most frequently executed basic blocks that together account for 95% of the execution time are considered hot. $T = 100\%$ corresponds with only the frozen code being considered cold. The figure also includes the size of the non-initialization code before cold code swapping is applied, and the size of the core code that must never be swapped out. For all data points in the graph, the cold

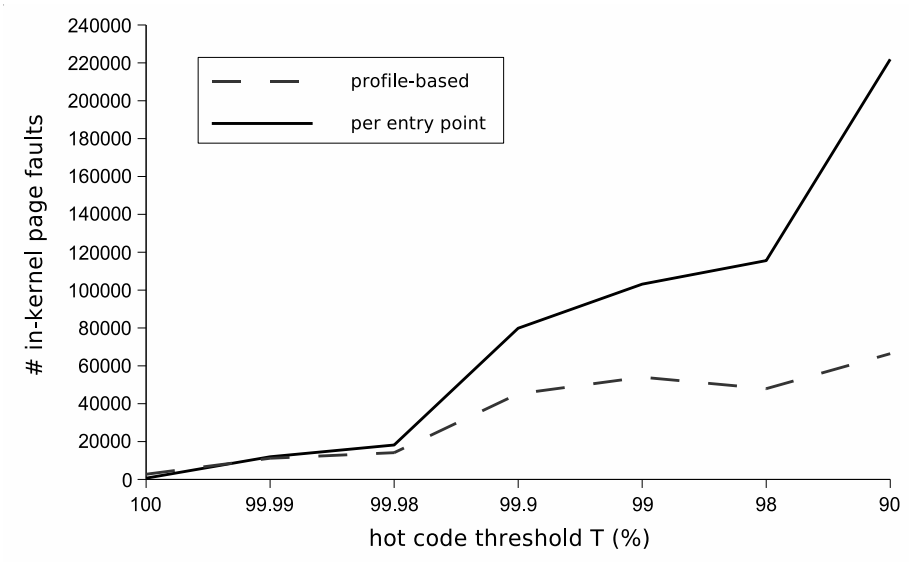


Figure 5.8: Number of kernel page faults in function of T for both cold code placement strategies.

code swapping was applied on top of all preceding link-time optimization and specialization transformations.

In line with our previous experience, slightly less than half of the remaining code after compaction and specialization is frozen. The remaining code size then drops off rapidly with decreasing T values: at $T = 100\%$, the code size is approximately 300 KiB, at $T = 99.9\%$ this has already diminished to 151 KiB. From this point onwards, the decrease happens much more slowly: at $T = 99\%$, the remaining code size is 136 KiB, at $T = 90\%$, 117 KiB of the non-initialization code remains, of which approximately 64 KiB is core code.

Clearly, the interesting range of T values lies between $T = 100\%$ and $T = 90\%$. Lower T values yield little to no extra gain, while they will substantially reduce the kernel's performance.

5.6.2 Evaluation of the Code Placement Strategies

For different T values, and for both cold code placement strategies, we have recorded the number of kernel page faults that occurred during five consecutive runs of the LMbench suite. In all cases, we have used

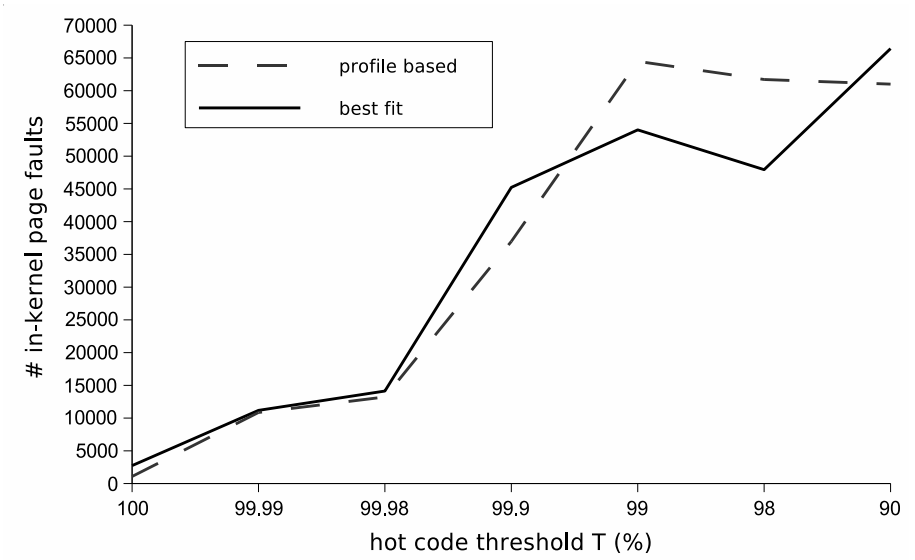


Figure 5.9: Number of kernel page faults in function of T for the profile-based placement strategy, with both small page merging strategies.

a 32-page swap-in buffer, the best-fit small pages merging algorithm and the NRU in-kernel page replacement policy. The results are shown in Figure 5.8. In order to make the results more clearly visible, the X axis of this graph does not have a linear scale. Instead all data points are spaced equidistantly.

The results confirm our intuition: for $T = 100\%$, the per entry point placement strategy gives better results than the profile-based placement strategy, although in both cases the total number of in-kernel page faults is of course low. As soon as the hot code threshold is lowered however, and useful profile information about the swapped-out code becomes available, the profile-based strategy outperforms the per entry point strategy. At $T = 99.99\%$, both strategies are almost tied (the profile-based strategy is already slightly better), but for lower T values the differences are very clear. While the number of page faults for the profile-based strategy hovers between 40000 and 60000 for low T values, the page faults for the per entry point strategy exceed 220000 for $T = 90\%$.

In order to quantify the influence of the small pages merging strategy on the number of page faults, we have done the measurements

from the previous graph for the profile-based placement strategy with the profile-based small pages merging strategy as well. The results are shown in Figure 5.9. In this case, there is no clear winner: while the profile-based merging strategy usually outperforms the best-fit strategy, it does so by a small margin, and not in all cases. Hence, we can conclude that the small pages merging strategy has little influence on the overall effectiveness of the cold code placement.

Table 5.4 shows the average performance degradation for the individual microbenchmarks of the LMbench suite for different T values. All measurements were done with the profile-based code placement and small pages merging strategies, with a 32-page swap-in buffer and the NRU page-replacement policy. The numbers indicate the relative performance degradation compared to the original, not rewritten kernel. Positive numbers indicate slowdowns, negative numbers indicate speedups. Boldface numbers indicate extreme slowdowns of more than 20%.

In general, for the higher T values ($T \geq 99.98\%$), no extreme slowdowns are noticeable. On average, there is a performance hit of approximately 1% for $T = 100\%$, and a performance improvement of approximately 2% for $T = 99.99\%$. This is slightly counterintuitive, as in the first case less code would need to be swapped in than in the second case. We suspect that the small performance loss caused by the extra swap-ins is in this case offset by the performance gained because of better I-cache utilization. Because the coldest code is moved to separate memory pages, the hot code is placed closer together, reducing the chance that hot cache lines are polluted with cold code.

For lower T values, there are big slowdowns for the *shell process* and *TCP connect* benchmarks. The cause of these big slowdowns is that in both cases a code path through the kernel that is uniquely exercised by these benchmarks is considered hot for high enough values of T, but this is no longer the case once T drops below a certain threshold. These (long) code paths are scattered over a number of cold pages, triggering several swap-ins. In the case of the *TCP connect* benchmark, for example, the original measured time was approximately 200 μs, whereas the time measured in the case of the extreme slowdown is in the order of 3000 μs. Given the page swap-in delay of 442 μs, this corresponds to approximately 6 pages being swapped in for one run of the benchmark. The effect is very pronounced because of the simple nature of the microbenchmark: it does nothing more than create a TCP connection

syscall and signal handling											*local communication latencies*			
T	null call	null I/O	stat	open/close	select TCP	sig inst	sig hndl	fork proc	exec proc	sh proc	pipe	AF UNIX	TCP	TCP conn
100.00%	6.06%	3.38%	0.55%	7.87%	6.98%	5.26%	-0.51%	1.64%	-0.44%	1.00%	4.88%	1.47%	-12.10%	-8.68%
99.99%	6.06%	3.38%	0.00%	1.01%	8.32%	4.78%	2.03%	0.54%	0.79%	1.11%	-5.80%	-2.73%	-15.46%	-11.26%
99.98%	4.04%	2.36%	1.64%	3.90%	6.17%	6.22%	0.00%	-0.18%	0.22%	1.41%	-5.80%	-9.22%	-18.40%	-9.28%
99.90%	6.06%	2.03%	0.73%	1.48%	0.36%	4.31%	-0.51%	-0.46%	0.50%	1.65%	-4.61%	-2.94%	-23.23%	**1333.64%**
99.00%	6.06%	3.38%	4.38%	-0.81%	3.49%	4.78%	-1.19%	1.23%	2.08%	**505.56%**	-4.71%	-8.81%	-23.49%	**998.17%**
98.00%	4.04%	2.70%	3.20%	1.62%	0.89%	4.63%	0.34%	1.38%	2.01%	**473.95%**	-6.89%	-9.22%	-23.89%	**1228.01%**
90.00%	0.00%	3.38%	0.46%	0.07%	1.43%	2.87%	-1.19%	2.11%	3.65%	**487.95%**	-4.21%	-1.47%	-10.52%	**1273.82%**

file & VM system latencies								*context switching*					
T	0K create	0K delete	10K create	10K delete	mmap latency	prot fault	page fault	2p/0K	2p/16K	2p/64K	8p/16K	8p/64K	16p/16K	16p/64K
100.00%	0.10%	-9.98%	6.60%	-9.19%	3.87%	-5.98%	0.00%	**26.78%**	1.03%	0.21%	3.64%	14.95%	1.77%	0.02%
99.99%	-1.63%	-20.29%	5.32%	-14.67%	1.68%	-17.70%	0.00%	-14.99%	-3.91%	-0.64%	0.81%	-2.46%	-12.54%	-1.54%
99.98%	-0.77%	-20.13%	6.40%	-16.32%	2.52%	16.79%	0.00%	1.23%	-2.26%	-0.28%	-0.81%	-3.45%	-8.85%	0.18%
99.90%	-0.53%	-12.42%	6.57%	-17.28%	2.18%	-12.42%	0.00%	-7.13%	-4.12%	-1.41%	-0.81%	-1.30%	-11.21%	-0.02%
99.00%	1.50%	-10.57%	7.34%	-15.15%	1.85%	-6.67%	0.00%	-5.65%	-2.88%	-1.27%	0.40%	2.90%	-9.59%	-0.04%
98.00%	1.70%	-13.61%	7.76%	-19.78%	3.19%	15.00%	0.00%	-7.62%	-3.91%	-1.91%	-1.01%	3.93%	-11.21%	-0.25%
90.00%	0.90%	-20.17%	8.51%	-12.84%	3.36%	-5.87%	0.00%	-6.39%	-1.44%	-0.64%	0.61%	1.13%	-5.90%	-0.31%

local communication bandwidths								
T	pipe	AF UNIX	TCP	file reread	mmap reread	bcopy (libc)	bcopy (hand)	mem read	mem write
100.00%	1.36%	-2.35%	-4.43%	0.27%	0.01%	-0.09%	-0.24%	0.00%	0.00%
99.99%	-0.15%	-3.13%	-3.92%	0.17%	-0.01%	-0.02%	-0.12%	0.00%	0.00%
99.98%	0.75%	-2.35%	-8.82%	-0.04%	0.12%	0.07%	0.02%	0.27%	0.00%
99.90%	-0.15%	-1.83%	-3.51%	-0.04%	0.13%	-0.02%	-0.07%	0.27%	0.00%
99.00%	1.06%	-3.39%	-2.70%	-0.02%	0.11%	0.02%	-0.05%	0.27%	0.00%
98.00%	2.11%	-0.52%	-0.81%	0.00%	0.13%	-0.21%	-0.26%	0.27%	0.05%
90.00%	0.60%	-2.35%	-2.74%	0.02%	0.00%	-0.25%	-0.26%	0.00%	0.00%

Table 5.4: Performance degradation per individual microbenchmark result.

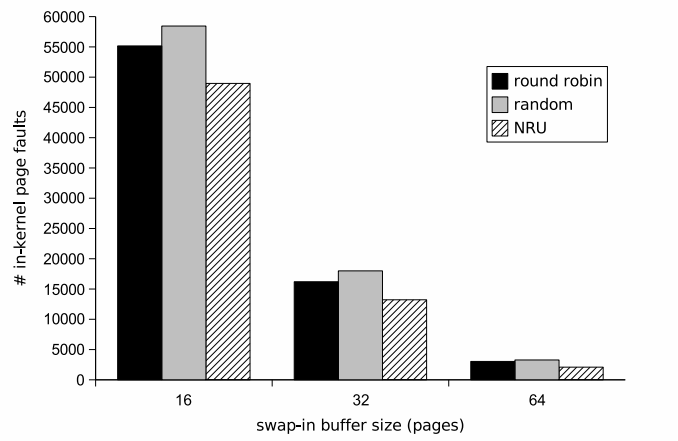


Figure 5.10: In-kernel page faults for different swap-in buffer sizes and replacement policies.

and then close it. Because the execution time of this benchmark is so small, even a single swap-in operation would cause the execution time to triple. Obviously, in real world programs, the effect of a low number of swap-ins would be relatively less noticeable when compared to the total execution time of the program.

5.6.3 Influence of Buffer Size and Replacement Policy

Next, we investigate the influence of the swap-in buffer size and in-kernel page replacement policy on the performance of the cold code swapping technique. For these experiments, we focused on a kernel rewritten with the profile-based code placement and small page merging strategies, and a fixed hot code threshold value $T = 99.98\%$. Buffer sizes are varied between 16, 32 and 64 pages, and three buffer management policies are considered: round robin, random replacement and NRU replacement.

Figure 5.10 shows the number of in-kernel page faults that occurred during five runs of the LMbench suite for each combination of swap-in buffer size and replacement policy. These results are entirely in line with our expectations. The size of the swap-in buffer has a severe impact on the total number of page faults: quadrupling the buffer size reduces the amount of page faults by a factor of 11. The impact of the

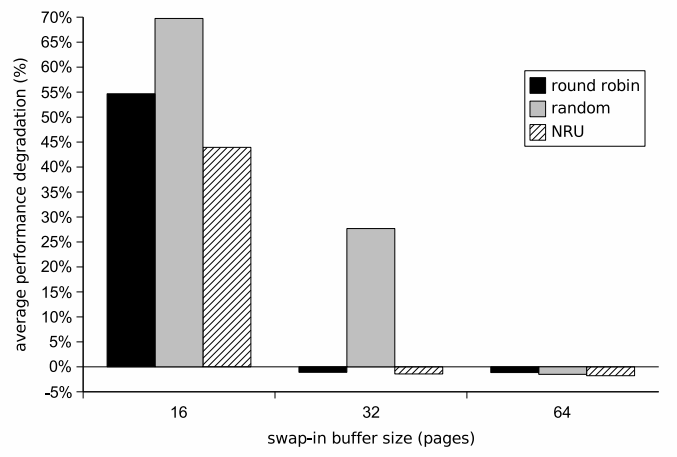


Figure 5.11: Average performance degradation for different swap-in buffer sizes and replacement policies.

page replacement policy is less pronounced, but it is clear that the NRU page replacement policy performs best, and random replacement consistently performs worst.

Figure 5.11 shows the average performance degradation for each combination of buffer size and replacement policy. The large average degradation for the 16-page buffer is due to only three microbenchmarks that experience extreme slowdowns that completely skew the average. The reason for this is that these microbenchmarks trigger a swap-in for more than 16 different pages, thus causing thrashing in the buffer. The same effect is noticeable for the 32-page swap-in buffer with random replacement policy. The randomness of the page replacement here causes the thrashing to occur even with a buffer that is large enough to hold all code needed for the microbenchmarks because still needed pages are already evicted from the buffer before all not-needed pages are.

Interestingly, for the round robin and NRU replacement policies there appears to be little performance difference between the 32-page and 64-page buffer cases. Hence, it seems that a buffer size of 32 pages is sufficiently large for practical purposes in this case study.

	text size	data size	init text size	init data size	image size	compressed image size	init mem. footprint	memory footprint
original kernel	714	292	46	8	932	470	1061	1006
+whole-program optimization	632 *-11.6%*	263 *-9.9%*	45 *-2.4%*	8 *-2.2%*	815 *-12.5%*	456 *-2.9%*	948 *-10.6%*	895 *-11.1%*
+specialization	563 *-21.2%*	244 *-16.5%*	31 *-32.2%*	9 *14.5%*	715 *-23.3%*	401 *-14.7%*	847 *-20.1%*	807 *-19.8%*
Cold Code Swapping								
+modif. page fault handler	568 *-20.5%*	248 *-15.2%*	33 *-28.4%*	10 *27.7%*	728 *-21.9%*	406 *-13.7%*	858 *-19.1%*	815 *-19.0%*
+swapping	148 *-79.3%*	248 *-15.2%*	33 *-28.4%*	10 *27.7%*	304 *-67.4%*	158 *-66.3%*	439 *-58.6%*	395 *-60.7%*
+init code duplication	142 *-80.2%*	248 *-15.2%*	39 *-14.8%*	10 *27.7%*	304 *-67.4%*	159 *-66.2%*	439 *-58.6%*	389 *-61.3%*
Including Overhead								
+buffer & repository	270 *-62.3%*	248 *-15.2%*	39 *-14.8%*	10 *27.7%*	804 *-13.8%*	659 *40.2%*	567 *-46.6%*	517 *-48.6%*

Table 5.5: Impact of cold code swapping on the kernel's memory footprint.

5.6.4 Impact on Memory Footprint

Returning to our embedded router case study, Table 5.5 shows the impact of cold code swapping on the kernel's memory footprint. In this case, the hot code threshold T was fixed at 99.98%, and the profile-based code placement and small pages merging strategies were used. The first three lines in the table recapitulate the memory footprint reductions achieved with the whole-program optimizations and specialization described in the previous chapters. The next line shows the overhead of the modified page fault handler code that was added to the kernel to support the cold code swapping technique. This overhead amounts to 5 KiB of code and 4 KiB of data. The next two lines show the results with cold code swapping enabled, both without and with initialization code duplication (see Section 5.5.2). The initialization code duplication optimization allows us to swap out 6 KiB of code that would otherwise have been considered core code. The last line includes a 32-page swap-in buffer and the size of the uncompressed cold code repository, in order to provide a realistic view of the final memory footprint reductions.

The final static RAM footprint reduction, taking into account a 32-page swap-in buffer, is 48.6%. This is almost double the final static RAM footprint reduction achieved with frozen code compression (26.3%). This is due to two different causes: first, the cold code swapping technique does not require the (compressed) unloaded code to be retained in memory, and second, the cold code swapping technique is not restricted to loading frozen code on demand.

The static ROM footprints, on the other hand, paint a less rosy picture. This was to be expected, as all swapped-out code is now stored in ROM in uncompressed form. In total, 421 KiB of code is swapped out. This code is placed on 125 pages, so the final size of the swapped-out code repository is 500 KiB. There are two reasons for the size increase of the swapped-out code:

- To separate the cold from the hot code, a number of direct jump instructions have been inserted to break up control flow dependencies between hot and cold code. Furthermore, because the distance in virtual memory between hot and cold code is very large, part of the already existing direct jump instructions that connected hot code to cold code had to be recoded with a four-byte offset instead of a one-byte offset.

- As the minimum granularity of the profile-based code placement strategy is a chain of basic blocks, there inevitably is some fragmentation. Pages will almost never be perfectly filled.

A possible extension to the cold code swapping technique could solve the problem of the drastically increased ROM footprint by storing the swapped-out code in compressed form in the Flash repository. An additional benefit of this approach for systems with slow Flash memory and fast processors would be increased performance: if the processor can decompress the loaded page fast enough, the decompression time may be completely compensated for by the reduction in load time.

In conclusion, we can state that the cold code swapping technique is very useful for reducing the kernel's static RAM footprint. If a good value for the hot code threshold T is chosen, there is only a limited performance impact, while the memory footprint reductions are substantial. The technique is, however, not suitable for reducing the kernel's ROM footprint. If it is possible to apply the cold code swapping technique (i.e., the processor supports virtual memory), this technique is preferable over the frozen code compression technique. In case the device does not offer the necessary hardware support for cold code swapping, frozen code compression can be a useful, albeit more limited, alternative.

Chapter 6

Related Work

In this chapter, we discuss related work. First, we give an overview of the most important program compaction and compression techniques. Then, we discuss OS kernel specialization and optimization. We conclude with an overview of the existing code placement techniques for the minimization of page fault occurrences.

6.1 Program Size Reduction Techniques

In general, there are two different approaches to program size reduction: compaction and compression. Compaction techniques produce a smaller program that is directly executable, while compression techniques require a decompression step (either in software or in hardware) before the program can be executed.

We now present an overview of the most important program compaction and compression techniques. A comprehensive overview can be found in [Besz03].

6.1.1 Compaction Techniques

A program's code size is dependent on the binary representation of its instructions. This binary representation is defined by the target processor's instruction set architecture (ISA). Traditional CISC processor architectures, like the i386, offered an instruction set that was engineered for code density. Complex but frequently occurring operations were represented by one instruction, and the variable-length in-

struction sets made it possible to assign a shorter representation to the most frequently occurring instructions and a longer one to those instructions that were used less. Most modern-day processor families, however, have a RISC ISA, with fixed-size instructions (usually 32 bit wide) that only encode relatively simple operations. Consequently, programs compiled for RISC processors typically have a larger code size than those compiled for CISC architectures. For at least two popular embedded RISC architectures, ARM and MIPS, this disadvantage has been countered by the introduction of an alternative instruction set, called Thumb and MIPS-16 respectively, whose instructions have a fixed width of 16 bit. The trade-off for the smaller instruction size is limited expressiveness: instead of three-address instructions (two source registers and one destination register), most instructions have a two-address form where the destination register is one of the source registers, not all registers can be used in all instructions and not all instructions from the original ISA have a 16-bit counterpart. Because of this, code compiled for the alternative ISA will comprise more, but smaller, instructions. Translating ARM code to Thumb instructions on average reduces the code size with 30% but the resulting code will execute on average 40% slower [Furb96]. Because programs can switch modes during execution, it is possible to compile performance-sensitive code for the 32 bit ISA and just translate the infrequently executed parts of the code to the alternative ISA. Krishnaswamy and Gupta [Kris02] explored this idea, and showed that the code size reduction is almost the same as when pure Thumb code is used, while the performance of the mixed ARM-Thumb code is nearly the same as that of the original pure ARM code.

Perhaps the most trivial compaction technique is compiler optimization. Most compiler optimizations not only speed up program execution but also reduce a program's footprint. The most notable exceptions to this rule are loop unrolling and procedure inlining. Most compilers offer the user the possibility to tune the optimization level, and even allow to optimize strictly for code size instead of execution speed (e.g., GCC's `-Os` command-line parameter). For embedded systems tool chains like ARM's RVCT, code size optimization is even the default behavior. A good overview of compile-time program optimization can be found in the book by Allen and Kennedy [Alle01]. As the different compiler optimizations interact with each other, it is necessary to find a good optimization ordering so that maximal code compaction is achieved. Cooper et al. [Coop99b] use a genetic algorithm to deter-

mine the best optimization sequence for either a set of programs or an individual program. They found that the ideal optimization sequence differs for each program, and report code size savings of up to 40% when the optimization sequence suggested by the genetic algorithm was used instead of the compiler's fixed optimization sequence.

However, compiler optimizations are hampered by the fact that a compiler only views one compilation unit (which usually corresponds to one source code file) at a time. Some optimizations like code factoring [Coop99a, DeSu02, Cheu03, Chen03b, DeSu05], which extracts similar code sequences into separate functions so they only need to occur once in the program, definitely benefit from a whole-program overview. This shortcoming can be overcome by moving some of the optimization burden from the compiler to a later stage of the tool chain, the linker. The linker's task is to join the separate compilation units and libraries in order to create one executable file. Traditionally, no optimization is done in this phase. However, it turns out that the link phase is well-suited for program size optimizations. By applying, amongst others, unreachable code and data elimination, code factoring at both procedure and basic block level, interprocedural constant propagation and liveness analysis, it is possible to reduce the static RAM footprint of statically linked programs for the Alpha architecture with on average 20% to 43% [DeSu05]. The Alpha architecture, however, is server-oriented and as such tool chains for these systems are not at all geared towards creating compact programs. Therefore, it makes more sense to evaluate the potential of link-time compaction on a real embedded architecture, with a tool chain that is more focused on generating small programs. We have studied link-time compaction of statically linked programs for the ARM architecture, compiled with the ARM Developer Suite and the ARM RealView Compiler Tools, which are known to produce very small binary executables. We have found that it is possible to achieve an average code size reduction of 14.6%, with an average reduction in execution time of 8.3%, while the total energy consumption decreased with 7.3% on average [DeSu07]. This proves the viability of link-time binary compaction for real embedded systems and shows that compaction can be done without compromising execution speed and energy consumption.

6.1.2 Compression Techniques

If the only optimization target is the ROM footprint, it is sufficient to use any known data compression technique to compress the complete program image and transform the program into a self-extracting executable. However, techniques have been developed that make use of the unique properties of program code to achieve a higher compression ratio. These are the so-called *wire codes*. They typically work by translating the code to some intermediate compiler format and applying compression to this representation. Ernst et al. [Erns97] use the intermediate code format of the LCC C compiler as a starting point for compression, Franz and Kistler [Fran97] use syntax trees. In both cases, decompression of the code involves some further compilation to achieve an executable representation. This makes these approaches less suitable for embedded systems, where the overhead of the further compilation would be too large. Drinić et al. [Drin03] propose a technique for compression of machine code based on prediction by partial matching (PPM). To improve the achieved compression ratio, they perform instruction rescheduling to improve the prediction probabilities for their compressor. For large i386 applications, they report compression ratios that are 18 to 24% better than those achievable with off-the-shelf compressors.

On some systems, compression is a hardware feature. Program code is then stored in a compressed format in memory — thus reducing both static RAM and ROM footprint — and decompressed by hardware. In one approach, decompression is done upon transfer from main memory to the processor cache. This has the advantage that the decompression is not in the critical path of the execution. This technique is used in the Compressed Code RISC Processor [Wolf92], where the code size of MIPS programs was reduced with 27%, and IBM's Codepack technology [Kemp98], which reduces the code size of PowerPC executables with on average 40%. The alternative is to perform decompression when the processor fetches the instructions from the cache. While this places the decompressor in the processor's critical path, and thus incurs some extra instruction fetch latency, it allows for a much better cache utilization. This in turn makes for a reduced power consumption as less accesses to main memory are needed. Xie et al. [Xie03] use profiling information to compress only those parts of the code that are infrequently executed to avoid the decompression latency as much as possible. They report good (but unspecified) compression ratios with a

performance penalty of only 6% compared to the execution of the uncompressed program.

Hardware compression of instructions is also often employed in VLIW architectures, where the wide instruction words often necessitate the inclusion of a large number of *no-op* operations in the instruction stream. These no-ops waste memory space and instruction fetch bandwidth, so they are often compressed in the memory representation of the program with no-op compression techniques [Colw88, Cont96, Adit00].

An alternative approach that also requires hardware support is the so-called *echo instruction*. Here, a dictionary is built that contains the most frequently occurring instruction sequences. Elsewhere in the code, these sequences are then replaced by a single echo instruction that points to an entry in the dictionary and specifies how many instructions from this entry need to be "echoed". This is in effect an LZ77 compression technique [Ziv77] adapted for use on program code, and can also be seen as a hardware-supported version of the procedural abstraction techniques used in program compaction. Fraser reports a code size reduction of on average 33% with this approach [Fras06].

There are several approaches that forego hardware decompression support and do the decompression in software. Kirovski et al. [Kiro97] propose a technique where whole procedures are decompressed at once. Upon each procedure call, the decompression buffer is checked for the presence of the called procedure. If it is not available, the decompressor will allocate space in the buffer and decompress the procedure. For SPARC code, on average a 40% compression is achieved, at the cost of a 10% slowdown.

Debray and Evans [Debr02] perform compression of more general code regions. In order to limit the performance impact, only cold code is compressed. If only the code that was never executed during the profiling runs is compressed, a code size reduction of 13.7% is achieved together with a small speedup in execution. By compressing code that is responsible for up to 0.005% of the execution time during profiling runs, a code size reduction of 18.8% was achieved, at the cost of a 27% execution slowdown. While these results seems worse than those achieved by Kirovski et al., they cannot be compared directly. The size reductions reported by Debray and Evans do not use the original executable as the baseline, but a version that has already been compacted with a link-time compactor. Furthermore, different ISAs were used

(SPARC versus Alpha) and different compression algorithms were applied. The frozen code compression technique described in this dissertation is very similar to the technique proposed in this work. The major difference is that Debray and Evans do not deal with concurrent programs, which makes it possible to easily evict decompressed code from memory. As an OS kernel is inherently multithreaded, code eviction is a lot harder for the frozen code compression technique, which is why we have chosen to omit it altogether.

Shogan and Childers [Shog04] propose a variation on the technique by Debray and Evans where decompression happens under the control of a *software dynamic translator* (SDT) instead of being integrated with the program itself. The SDT is a software layer that sits between the program and the operating system. Compressed code regions are fetched by the SDT and decompressed into a buffer before execution. When the control flow exits the decompressed region it returns to the SDT, which then decompresses the next region, evicting previously decompressed regions from the buffer as needed. The authors report code size reductions of about 45%.

Ozturk et al. [Oztu05] compress the complete program code and do decompression with basic block granularity. To offset the performance loss caused by software decompression, they use a predictive decompression strategy that decompresses basic blocks that are expected to be executed ahead of time in a separate decompression thread. A third thread handles eviction of decompressed basic blocks when they are not expected to be executed in the near future. Unfortunately, no code size savings or performance measurements are mentioned in the article.

Waldman and Pinter [Wald06] use a code compression technique to reduce a program's static ROM footprint. Code size reduction and runtime overhead are traded off based on profile information. The code regions selected for compression are single-entry, single-exit regions that vary in size from individual basic blocks to whole procedures. The decompression buffers are dynamically allocated at run time, and there is no predetermined upper bound on the total amount of memory used for decompression buffers. The authors report an average code size reduction of 18.5%, at the cost of a 3.8% increase in memory usage at run time and a 7.8% increase in execution time.

Citron et al. [Citr04] propose to replace compression with delayed code and data loading. Through profiling analysis, the *frozen* code and data of a program are identified. These are the code and data frag-

ments that are never executed or referenced during the profiling runs. The frozen code and data are then removed from the program and replaced with stubs that load them from disk or ROM if they are needed during execution. For the mediabench benchmark suite [Lee97], which is representative for embedded programs, an average static RAM footprint reduction of 78% is achieved, while less than 1% of the code and data needs to be loaded during execution. This technique allows for reduction of a program's static RAM footprint, but it does not reduce the ROM footprint, as the frozen code and data are still stored in ROM. An extra advantage of this technique is that application startup times are significantly reduced because less than one quarter of the original program needs to be loaded at program startup.

An alternative take on the compression idea refers back to the observation that a program's image size depends on the ISA it is compiled for. By defining an application-specific instruction set and providing an interpreter for this ISA, the code size can be reduced if the size of the native code is larger than the size of the interpreted program and the native interpreter combined. With this approach Fraser and Proebsting [Fras95, Proe95] are able to reduce the code size of C programs by about 50%. A variation on this technique is used by Hoogerbrugge et al. [Hoog99] to compress code for the TriMedia architecture. In order to limit the performance loss they only compress cold code. The interpreted language's instruction set is constructed based on a set of training applications, and is not specific to the one program being compressed. This approach removes about 80% of the code size and results in an eightfold execution slowdown. Clausen et al. [Clau00] extend the Java bytecode language with application-specific macro-instructions that replace frequently occurring bytecode sequences. A modified Java Virtual Machine can then interpret the macro-instructions and execute the program. This approach results in an average code size reduction of 15%.

6.2 OS Memory Footprint Reduction

The idea of specializing the Linux kernel for a specific application was first explored by Lee et al. [Lee04]. Based on source code analysis, a system-wide call graph is built that spans the application, the libraries and the kernel. On this graph, a reachability analysis is performed, resulting in a compaction of the kernel of 17% in a simple case study.

We believe our approach to be more general, as it is source-language independent, and because more optimizations can be performed at link time.

He et al. [He07] use an approach similar to ours to reduce the code size of the Linux kernel. They use a different link-time binary rewriting framework however, PLTO [Schw01], that only supports the i386 architecture. They only apply the standard whole-program link-time optimizations and two specializations: system call elimination and boot process specialization. A novelty in their approach is the use of *approximate decompilation* to generate C source code for hand-written assembly code in the kernel. This allows the use of a source code based pointer analysis (the FA-analysis [Mila04]) for the identification of targets of indirect function calls. While the generated source code is not functionally equivalent to the original assembler code, it exhibits the same properties with regards to this FA-analysis. On a Linux 2.4 kernel without networking support, they report a code size reduction of 23.83%. For the same test system we have used for an earlier evaluation of the specialization techniques presented in Chapter 4 [Chan05], their results are similar to ours, suggesting the two techniques are approximately equal in strength.

In an earlier paper by the same research group, Rajagopalan et al. [Raja06] describe the challenges in rewriting and instrumenting a Linux kernel with PLTO. The problems they faced were mostly the same as those described in Chapter 3.

An alternative approach to customize an OS for use in embedded devices is proposed by Bhatia et al. [Bhat04]. Instead of manually customizing the OS for specific hardware features and handcrafting the generic code base to a hardware specific version, the authors of this paper propose to remotely customize OS modules on demand. A customization server provides a highly optimized and specialized version of an OS function on demand of an application. The embedded device needs to send the customization context and the required function to the server and on receipt of the customized version, applications can start using it. The size of the customized code is reduced by up to a factor of 20 for a TCP/IP stack implementation for ARM Linux, while the code runs 25% faster and throughput increases by up to 21%.

While our approach to minimizing the kernel's memory footprint is top-down in that we start with a full-featured kernel and strip away as much unneeded functionality as possible, there are a number of

projects that take a bottom-up approach. The Flux OSKit [Ford97], Think [Fass02] and TinyOS [Gay03] are operating system building frameworks that offer a library of system components to the developer, allowing him to assemble an operating system kernel containing only the needed functionality for the system.

McNamee et al. [McNa01] introduce a toolkit for the systematic specialization of operating system code. Their approach is based on the partial evaluation of source code, generating specialized versions of parts of the kernel. The specialization can happen both statically, at system build time, and dynamically, at run time, but only the static approach is really useful with regards to memory footprint reduction. The focus of this work is also more on achieving performance improvements than on kernel memory footprint reduction.

6.3 Other OS Kernel Optimization Approaches

Krintz and Wolski [Krin05] propose to apply specialization to the Linux kernel to improve the performance of scientific applications in batch processing systems. In such systems, only one application is running at any time, so it is possible to tailor the kernel to this one application. For every new job that is run on the system, a new, specialized kernel is loaded. The objective of this work is to improve the performance of the scientific application, not to reduce the memory footprint of the kernel.

Spike [Cohn97] is a (post-)link-time optimizer for the Alpha architecture that includes a profile-guided code layout optimization to improve cache usage. Spike has also been used to optimize Tru64Unix kernels [Flow01] for speed, both through profile-guided code layout and through the profile-guided insertion of data prefetching instructions. Performance improvements of up to 40% on a set of benchmarks running on an optimized kernel were reported for this Spike version.

Perianayagam et al. [Peri06] use PLTO to instrument a Linux kernel to collect basic block and edge profiles, call trace profiles and system call traces for a representative set of applications. This information is then used to specialize the kernel for these applications: new system calls are created that are specialized for frequently occurring system call parameters, aggressive procedure inlining is applied to optimize the most frequently executed code paths and profile-guided code layout optimizations are applied. The achieved performance gains are, however, very small.

In contrast with link-time optimization, most kernels are traditionally optimized by the compiler only. To detect kernel bottlenecks, profile information is used. Profile-guided restructuring of the operating system for the optimization of its throughput or latency has been studied for AS400 [Schm98] and HP-UX [Spee94] platforms.

The KernInst dynamic kernel instrumentation system [Tamc99] has been used to optimize parts of the UltraSPARC Solaris kernel [Tamc01]. As its approach is dynamic, it cannot optimize for code size: the entire kernel has to remain in memory. KernInst uses a code positioning scheme, similar to the one used by Spike, which results in speedups up to 17.6% for selected functions.

In the past, there have been many research projects that focused on dynamically specializing OS kernel subsystems in order to improve the performance of specific applications. A comprehensive overview of the varied approaches is given by Denys et al. [Deny02]. None of these approaches is particularly useful in reducing the memory footprint of a kernel, however.

6.4 Code Reordering for Page Fault Minimization

Hatfield and Gerald [Hatf71] describe a technique that aims to minimize the number of page faults for both code and data references. The code and data are divided into a set of relocatable blocks (e.g., an array or a procedure). Using profile data, a *nearness matrix* is constructed, with one row and column for each relocatable block. Entry c_{ij} of this matrix represents the count of references from block i to block j. Virtual memory pages correspond to square regions along the diagonal of the matrix. By reordering the rows and columns of the matrix, the largest entries are brought closest to the diagonal, which corresponds to placing the blocks that reference each other most on the same page.

Ferrari [Ferr74] formulates the problem as a graph clustering problem. Nodes in the graph represent relocatable blocks. The weight of a node equals the size of the block it represents. Edges in the graph represent interblock references, and can be weighted according to various cost functions. An optimal ordering is then sought by clustering graph nodes in such a way that no node becomes larger than the page size and the total weight of the remaining edges is minimal. If the edges are weighted according to profile information, this method is equivalent to that of Hatfield and Gerald. The author proposes a better-performing

edge weighting, however, that is based on a trace of the block references during execution instead of mere profile data.

Pettis and Hansen [Pett90] propose to reorder the procedures in a program in such a way that those procedures that call each other most frequently are placed closest together. Their main aim is to reduce the number of conflict misses in the instruction cache, but they note that this placement algorithm also reduces the number of page faults during program execution. Once again, the program is represented as a graph, with the nodes representing the procedures, and edges representing procedure calls. The edges are weighted according to profile information. In each step of the algorithm, the edge with the highest weight is selected and its head and tail nodes are merged. This method does not prevent procedures from spanning page boundaries.

Gloy and Smith [Gloy99] also reorder procedures to improve a program's instruction memory hierarchy behavior. Their technique is similar to Pettis and Hansen's, but instead of profile information they use *temporal ordering information*, which not only summarizes the number of calls from one procedure to another, but also in which way these calls are interleaved. While their approach is in the first place directed towards optimization of the cache utilization, the authors also discuss an extension of the technique to minimize the amount of page faults.

Chapter 7

Conclusions and Future Work

In this dissertation, we have shown how the RAM and ROM memory footprint of an OS kernel can be reduced through the application of link-time binary rewriting techniques. We have identified the challenges OS kernel code presents to a link-time rewriter, and we have shown how these challenges can be overcome. As an extension to the standard link-time compaction optimizations, we have proposed a number of specializations that streamline the kernel for a specific hardware/software combination. Finally, we have introduced two approaches to on-demand code loading that allow to reduce the kernel's footprint even further, without significantly decreasing its performance.

7.1 Conclusions

Table 7.1 gives an overview of the memory footprint reductions that were achieved with the techniques presented in this work.

The combined optimization and specialization techniques result in a 19.8% reduction of the static memory footprint after initialization for the i386 kernel, and a 17.6% reduction for the ARM kernel. The uncompressed image size is reduced by 23.3% and 22.1% respectively, while the compressed image size is reduced by 14.7% and 9.5% respectively.

The introduction of on-demand code loading in the kernel allows us to reduce the static RAM footprint after initialization even more.

	text size	data size	init text size	init data size	image size	compressed image size	init mem. footprint	memory footprint
original kernel	714	292	46	8	932	470	1061	1006
+whole-program optimization	632 *-11.6%*	263 *-9.9%*	45 *-2.4%*	8 *-2.2%*	815 *-12.5%*	456 *-2.9%*	948 *-10.6%*	895 *-11.1%*
+specialization	563 *-21.2%*	244 *-16.5%*	31 *-32.2%*	9 *14.5%*	715 *-23.3%*	401 *-14.7%*	847 *-20.1%*	807 *-19.8%*
Frozen Code Compression								
+frozen code compression	276 *-61.4%*	466 *59.5%*	31 *-32.2%*	9 *14.5%*	652 *-30.1%*	447 *-5.0%*	782 *-26.2%*	742 *-26.3%*
Cold Code Swapping								
+cold code swapping	270 *-62.3%*	248 *-15.2%*	39 *-14.8%*	10 *27.7%*	804 *-13.8%*	659 *40.2%*	567 *-46.6%*	517 *-48.6%*

(a) i386

original kernel	1060	203	47	4	1207	575	1315	1264
+whole-program optimization	935 *-11.8%*	203 *-0.2%*	44 *-7.2%*	4 *0.0%*	1041 *-13.7%*	574 *-0.1%*	1185 *-9.8%*	1138 *-10.0%*
+specialization	839 *-20.8%*	202 *-0.6%*	39 *-17.4%*	4 *-2.7%*	940 *-22.1%*	520 *-9.5%*	1084 *-17.5%*	1042 *-17.6%*
Frozen Code Compression								
+frozen code compression	367 *-65.3%*	533 *161.8%*	40 *-16.3%*	4 *-2.7%*	797 *-34.0%*	557 *-3.0%*	943 *-28.2%*	900 *-28.8%*

(b) ARM

Table 7.1: Overall impact of the optimization and specialization transformations and the code loading techniques on the kernel's memory footprint.

Frozen code compression results in a reduction of 26.3% for the i386, and 28.8% for the ARM. Cold code swapping for the i386 kernel even achieves a 48.6% reduction of the footprint after initialization. The on-demand code loading techniques are, however, less suitable for reduction of the kernel's ROM footprint: frozen code compression reduces the gains for the compressed image size to 5% and 3% for the i386 and ARM kernels respectively. Cold code swapping even increases the compressed image size with 40% compared to the original i386 kernel. This size increase can be avoided by storing the swapped-out pages in compressed form in ROM.

The performance impact of the proposed transformations is minimal: by judiciously choosing the optimization parameters (especially in the case of cold code swapping), the performance of the rewritten kernel lies within 1 to 2% of that of the original kernel.

We can conclude that the usefulness of the proposed transformations is dependent on the ultimate optimization target. If reducing the RAM usage is the main concern, all optimizations should be applied, and cold code swapping should be preferred over frozen code compression if the hardware platform allows to do so. If reducing the ROM footprint is more important, it is advisable not to introduce code loading, and for maximal reduction of the compressed image size duplicate basic block elimination should not be applied either.

In general, the techniques presented in this dissertation allow embedded system designers to significantly reduce the OS kernel's memory footprint without compromising its performance. The proposed techniques all operate automatically or semi-automatically, requiring little human intervention or knowledge of the kernel's internal workings. The static RAM footprint reduction of up to 48% for the general-purpose Linux kernel makes it more feasible to use such a general-purpose OS for embedded systems, which has the potential of significantly reducing the design time for a device.

7.2 Future Work

In this section, we take a look at possible extensions to the work presented in this dissertation. We discuss both a number of direct extensions, and we also look somewhat further ahead, towards the ideal of whole-system optimization.

7.2.1 Direct Extensions

There are a number of direct, obvious extensions possible to the work presented in this dissertation. On the one hand, there are still more specialization opportunities in OS kernels that have not yet been fully explored. For example, the Linux kernel exposes a lot of internal information to user space programs as virtual files in the `/proc` directory tree. Currently, the `proc` interface is an all-or-nothing proposition: if it is enabled, all kernel subsystems and drivers expose their information in the directory tree. It would be better if the interfaces could be enabled selectively: through an analysis of the user space programs, the specializer could draw up a list of all `/proc` interfaces that are in effect used by the software. The code that exposes all other, unused information could then be discarded from the kernel.

As part of the kernel command line specialization, the kernel code is specialized for known constant values of configuration variables. This technique could be applied to other, non-configuration, variables as well if these are known to be constant for a particular system. For example, some variables in hardware drivers are assigned a value based on the results of hardware probing in the bootup phase of the system. Assuming the system's hardware configuration does not change over time, these variables will always be assigned the same value. Hence, a valuable extension to the current work would be a systematic method for discovering such system-dependent constants, so that the kernel code can be specialized for them.

The on-demand code loading schemes presented in this work can also be extended to include on-demand loading of (read-only) data. The main challenge here is determining which data to load on demand, and when to load it, as it is a lot more unwieldy to instrument all of the kernel's data accesses than it is to instrument the kernel code for the collection of basic block profiles.

As the work presented in this dissertation focuses on reducing the kernel's static RAM and ROM footprint, an obvious extension is the incorporation of techniques that reduce the dynamic RAM footprint of the kernel, such as the heap compression technique proposed by Zhang and Gupta [Zhan06b].

7.2.2 Looking Further Ahead

In the longer term, it should be possible to optimize all system components at once. As systems become more complex, designers will turn increasingly to pre-built components that together make up the system's software. The overhead this entails can, at least partially, be contained by the introduction of an automated whole-system integration step that streamlines the individual components by removing unused functionalities, and optimizes the communication between components by integrating them more tightly, perhaps even moving code and/or data from one component (OS, library, program, ...) to another. This vision has already been put forward by Rajagopalan et al. [Raja03] in 2003. The work presented in this dissertation allows for a correct modeling of the OS kernel as part of the whole system, and as such allows for including the OS in this integration step.

One of the most important challenges for this research direction will be devising a proper, unified model for the intercomponent communications. In this dissertation, we have only touched lightly on this subject: the only real intercomponent communications we considered were system calls, which can easily be modeled as something akin to regular function calls on the binary instruction level. However, there are more complex ways of interaction between components possible as well: interprocess communication through sockets or shared memory, remote procedure calls for distributed systems, communication with the OS kernel through virtual file systems such as the aforementioned `proc` filesystem come to mind.

When the system integration happens at the binary level, these communication channels appear as a succession of abstraction layers and data passed into them will inevitably be lost somewhere in the complexities of the system before it can be propagated to the other end, where it is actually consumed. For example, a program that consumes kernel data through the `proc` interface has to open the relevant virtual file with a regular `open` system call, and read the information it needs with a regular `read` system call. In the kernel, the `read` system call handler delegates the work to the `proc` subsystem, where it is then further delegated to the specific handler installed for the `proc` entry that is read. This process involves at least two indirect procedure calls, and possibly a number of argument conversions.

A good modeling of the data flow through the different system components can avoid this data loss problem. By defining the start and end

points of each communication channel, and describing how data is converted upon transfer through the channel, it will be possible to perform effective intercomponent data flow analysis.

Appendix A

Gathering Profile Information

Basic block profiles for the kernel are generated by instrumenting the kernel with a modified version of our link-time rewriter. The instrumentation added to the kernel is very straightforward: an extra zero-initialized data section that contains a counter for each basic block in the kernel is inserted, and in each basic block the necessary instructions are added to increment that block's counter. Obviously, to preserve the correct working of the kernel, no live register values may be corrupted by the instrumentation code.

On the i386, the counter can be incremented with one instruction, `inc $counter`. While the `inc` instruction does not overwrite any registers, it does update the processor's condition flags, so if the original value of these flags is still needed later on, it needs to be saved and restored by adding a `pushf` instruction before and a `popf` instruction after the `inc`.

On the ARM architecture, the most concise way to increment a basic block's counter requires at least two free registers: one in which the address of the counter is produced, and another to load the counter value in so that it can be updated. The standard interprocedural liveness analysis of our link-time rewriter is used to identify two registers containing dead values. If less than two such registers can be found, spill code is inserted to save and restore the values of the used registers on the run-time stack.

There are no special accommodations for reading out the counter

values. The Linux kernel already offers the possibility to access the contents of the kernel's memory through the `/proc/kcore` interface, so the counter values are read directly from this interface.

Using the collected basic block profile information, our link-time rewriter then classifies the kernel code in two categories, *hot* and *cold* code, based on a user-configurable threshold value T. For example, for $T = 0.95$, the most-executed basic blocks that together constitute (approximately) 95% of the kernel's execution time will be considered hot. The hot code is identified with the following algorithm:

1. Compute the control flow graph's total weight. The total weight is defined as $W = \sum_{i=1}^{n} weight(block_i)$, where n is the number of basic blocks in the graph, $weight(block_i)$ is the execution count of the ith block multiplied by its number of instructions.

2. Sort the basic blocks on execution count in descending order.

3. Walk the sorted block list, summing the block weights until the accumulated weight is higher than or equal to $T * W$.

4. The control flow graph's *hot threshold count* H is then equal to the execution count of the last-visited block. All blocks whose execution count is higher than or equal to H are considered hot, all other blocks are cold.

Note that this algorithm is not exact, though the approximation it provides is sufficiently accurate to be useful.

Bibliography

[Adit00] S. Aditya, S. Mahlke, and B. Rau. Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):752–773, 2000.

[Alle01] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2001.

[Anck04] B. Anckaert, B. De Sutter, D. Chanet, and K. De Bosschere. Steganography for executables and code transformation signatures. In *Proceedings of Information Security and Cryptology - ICISC 2004*, pp. 425–439, 2004.

[Bans06] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 394–403, New York, NY, USA, 2006. ACM Press.

[Bert06] R. Bertran, M. Gil, J. Cabezas, V. Jimenez, L. Vilanova, E. Morancho, and N. Navarro. Building a global system view for optimization purposes. In *Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture at International Symposium on Computer Architecture*, June 2006.

[Besz03] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.

[Bhat04] S. Bhatia, C. Consel, and C. Pu. Remote customization of systems code for embedded devices. In *EMSOFT '04: Pro-*

ceedings of the 4th ACM international conference on Embedded software, pp. 7–15, New York, NY, USA, 2004. ACM Press.

[Chan05] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the Linux kernel. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp. 95–104, New York, NY, USA, 2005. ACM Press.

[Chan07a] D. Chanet, J. Cabezas, E. Morancho, N. Navarro, and K. De Bosschere. Linux kernel compaction Through cold code swapping. *Transactions on HiPEAC*, 2(2):60–88, 2007.

[Chan07b] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Automated reduction of the memory footprint of the Linux kernel. *Trans. on Embedded Computing Sys.*, 6(4), 2007. To appear.

[Chen03a] G. Chen, M. Kandemir, N. Vijaykrishnan, M. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 282–301, New York, NY, USA, 2003. ACM Press.

[Chen03b] W. Chen, B. Li, and R. Gupta. Code compaction of matching single-entry multiple-exit regions. In *Static Analysis: 10th International Symposium (SAS)*, volume 2694/2003 of *Lecture Notes in Computer Science*, pp. 401–417, June 2003.

[Cheu03] W. Cheung, W. Evans, and J. Moses. Predicated instructions for code compaction. In *Proceedings of Software and Compilers for Embedded Systems (SCOPES)*, volume 2826/2003 of *Lecture Notes in Computer Science*, pp. 17–32. Springer Berlin / Heidelberg, 2003.

[Citr04] D. Citron, G. Haber, and R. Levin. Reducing program image size by extracting frozen code and data. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pp. 297–305, New York, NY, USA, 2004. ACM Press.

[Clau00] L. Clausen, U. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489, 2000.

[Cohn97] R. Cohn, D. Goodwin, P. Lowney, and N. Rubin. Spike: an optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*, pp. 17–24, 1997.

[Colw88] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):967–979, 1988.

[Cont96] T. Conte, S. Banerjia, S. Larin, K. Menezes, and S. Sathaye. Instruction fetch mechanisms for VLIW architectures with compressedencodings. *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pp. 201–211, 1996.

[Coop99a] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp. 139–149, New York, NY, USA, 1999. ACM Press.

[Coop99b] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, July 1999.

[Cour04] J. Cours, N. Navarro, and W. Hwu. Using coverage-based analysis to automate the customization of the Linux kernel for embedded applications. Master's Thesis, University of Illinois at Urbana-Champaign, 2004.

[Debr98] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proceedings of the ACM 1998 Symposium on Principles of Programming Languages (POPL'98)*, pp. 12–24, 1998.

[Debr00] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[Debr02] S. Debray and W. Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 95–105, New York, NY, USA, 2002. ACM Press.

[DeBu03] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter. Post-pass compaction techniques. *Commun. ACM*, 46(8):41–46, 2003.

[DeBu04a] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, and K. De Bosschere. The design and implementation of FIT: a flexible instrumentation toolkit. In *PASTE '04: Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 29–34, New York, NY, USA, 2004. ACM Press.

[DeBu04b] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp. 211–220, New York, NY, USA, 2004. ACM Press.

[DeBu05] B. De Bus. *Reliable, retargetable and extensible link-time program rewriting*. PhD Dissertation, Ghent University, 2005.

[Deny02] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Comput. Surv.*, 34(4):450–468, 2002.

[DeSu01] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray. Combining global code and data compaction. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 29–38, 2001.

[DeSu02] B. De Sutter, B. De Bus, and K. De Bosschere. Sifting out the mud: low level C++ code reuse. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 275–291, 2002.

[DeSu05] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, September 2005.

[DeSu06] B. De Sutter, B. De Bus, and K. De Bosschere. Bidirectional liveness analysis, or how less than half of the alpha's registers are used. *J. Syst. Archit.*, 52(10):535–548, 2006.

[DeSu07] B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere. Link-time compaction and optimization of ARM executables. *Trans. on Embedded Computing Sys.*, 6(1):5, 2007.

[Drin03] M. Drinić, D. Kirovski, and H. Vo. Code optimization for code compression. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pp. 315–324, Washington, DC, USA, 2003. IEEE Computer Society.

[Erns97] J. Ernst, W. Evans, C. Fraser, T. Proebsting, and S. Lucco. Code compression. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pp. 358–365, New York, NY, USA, 1997. ACM Press.

[Fass02] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: a software framework for component-based operating system kernels. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pp. 73–86, Berkeley, CA, USA, 2002. USENIX Association.

[Ferr74] D. Ferrari. Improving locality by critical working sets. *Commun. ACM*, 17(11):614–620, 1974.

[Flow01] R. Flower, C. Luk, R. Muth, H. Patil, J. Shakshober, R. Cohn, and G. Lowney. Kernel optimizations and prefetch with the Spike executable optimizer. In *Proc of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[Ford97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for kernel and language research. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pp. 38–51, New York, NY, USA, 1997. ACM Press.

[Fran97] M. Franz and T. Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[Fras95] C. Fraser and T. Proebsting. Custom instruction sets for code compression. Technical report, Microsoft Research, 1995.

[Fras06] C. Fraser. An instruction for direct interpretation of LZ77-compressed programs. *Software: Practice and Experience*, 36(4):397–411, 2006.

[Furb96] S. Furber. *ARM system architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[Gay03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: a holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 1–11, New York, NY, USA, 2003. ACM Press.

[Gilb06] J. Gilbert and D. Abrahamson. Adaptive object code compression. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 282–292, New York, NY, USA, 2006. ACM Press.

[Gloy99] N. Gloy and M. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, 1999.

[Gupt05] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 263–272, New York, NY, USA, 2005. ACM Press.

[Hatf71] D. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[He07] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews. Code compaction of an operating system kernel. In *CGO '07: Proceedings of the international symposium on Code Generation and Optimization*, pp. 283–298. IEEE Computer Society, March 2007.

[Henn03] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach, 3rd edition*, Chapter 5. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[Hind01] M. Hind. Pointer analysis: haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pp. 54–61, New York, NY, USA, 2001. ACM Press.

[Hoog99] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. Wiel. A code compression system based on pipelined interpreters. *Softw. Pract. Exper.*, 29(11):1005–2023, 1999.

[Hu06] W. Hu, J. Hiser, D. Williams, A. Filipi, J. Davidson, D. Evans, J. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pp. 2–12, New York, NY, USA, 2006. ACM Press.

[Kemp98] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach. A decompression core for PowerPC. *IBM J. Res. Dev.*, 42(6):807–812, 1998.

[Kiro97] D. Kirovski, J. Kin, and W. Mangione-Smith. Procedure based program compression. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 204–213, Washington, DC, USA, 1997. IEEE Computer Society.

[Krin05] C. Krintz and R. Wolski. Using phase behavior in scientific application to guide Linux operating system customization. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10*, p. 219.1, Washington, DC, USA, 2005. IEEE Computer Society.

[Kris02] A. Krishnaswamy and R. Gupta. Profile guided selection of ARM and thumb instructions. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pp. 56–64, New York, NY, USA, 2002. ACM Press.

[Latt05] C. Lattner and V. Adve. Transparent pointer compression for linked data structures. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pp. 24–35, New York, NY, USA, 2005. ACM Press.

[Lee97] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. *Microarchitecture, 1997. Proceedings. Thirtieth Annual IEEE/ACM International Symposium on*, pp. 330–335, 1997.

[Lee04] C. Lee, J. Lin, Z. Hong, and W. Lee. An application-oriented Linux kernel customization for embedded systems. *Journal of Information Science and Engineering*, 20(6):1093–1107, 2004.

[Levi00] J. Levine. *Linkers & loaders*. Morgan Kaufmann Publishers, 2000.

[Linn] C. Linn, S. Debray, G. Andrews, and B. Schwarz. Stack analysis of x86 executables. Available from `http://www.cs.arizona.edu/people/debray`.

[Mado04] M. Madou, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Link-time optimization of MIPS programs. In *Proceedings of the 2004 International Conference on Embedded Systems and Applications (ESA'04)*, pp. 70–75, 2004.

[Mado06] M. Madou, L. Put, and K. Bosschere. Understanding Obfuscated Code. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 268–274, Washington, DC, USA, 2006. IEEE Computer Society.

[McNa01] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Trans. Comput. Syst.*, 19(2):217–251, 2001.

[McVo96] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[Mila04] A. Milanova, A. Rountev, and B. Ryder. Precise call graphs for C programs with function pointers. *Automated Software Engg.*, 11(1):7–26, 2004.

[Much97] S. Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann Publishers, 1997.

[Muth01] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. alto: a link-time optimizer for the Compaq Alpha. *Software - Practice and Experience*, 31(1):67–101, 2001.

[Nohr93] M. Nohr. *UNIX System V: understanding ELF object files and debugging tools*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1993.

[Oztu05] O. Ozturk, H. Saputra, M. Kandemir, and I. Kolcu. Access pattern-based code compression for memory-constrained embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 882–887, Washington, DC, USA, 2005. IEEE Computer Society.

[Peri06] S. Perianayagam, H. He, M. Rajagopalan, G. Andrews, and S. Debray. Profile-guided specialization of an operating system kernel. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2006.

[Pett90] K. Pettis and R. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pp. 16–27, New York, NY, USA, 1990. ACM Press.

[Proe95] T. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 322–332, New York, NY, USA, 1995. ACM Press.

[Raja03] M. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. Cassyopia: compiler assisted system optimization. In *Proceedings of the 9th Usenix Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[Raja06] M. Rajagopalan, S. Perianayagam, H. He, G. Andrews, and S. Debray. Binary rewriting and instrumentation of an operating system kernel. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2006.

[Schm98] W. Schmidt, R. Roediger, C. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky. Profile-directed restructuring of operating system code. *IBM Systems Journal*, 37(2), 1998.

[Schw01] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: a link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, 2001.

[Shog04] S. Shogan and B. Childers. Compact binaries with code compression in a software dynamic translator. In *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 1052–1059, Washington, DC, USA, 2004. IEEE Computer Society.

[Spee94] S. Speer, R. Kumar, and C. Partridge. Improving UNIX kernel performance using profile based optimization. In *1994 Winter USENIX*, pp. 181–188, 1994.

[Sriv94] A. Srivastava and D. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proc. of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 49–60, 1994.

[Tamc99] A. Tamches and B. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pp. 117–130, 1999.

[Tamc01] A. Tamches and B. Miller. Dynamic kernel code optimization. In *Workshop on Binary Translation (WBT-2001)*, 2001.

[Turl05] J. Turley. Embedded systems design survey: operating systems up for grabs. *Embedded Systems Design*, May 2005.

[Turl06] J. Turley. Embedded systems design survey: operating systems on the rise. *Embedded Systems Design*, June 2006.

[VanP05a] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pp. 7–12, December 2005.

[VanP05b] L. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere. LANCET: a nifty code editing tool. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 75–81, New York, NY, USA, 2005. ACM Press.

[VanP07a] L. Van Put. *Advanced link-time program analysis*. PhD Dissertation, Ghent University, 2007.

[VanP07b] L. Van Put, D. Chanet, and K. De Bosschere. Whole-program linear-constant analysis with applications to link-time optimization. In *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems*, pp. 61–70, 2007.

[Wald06] I. Waldman and S. Pinter. Profile-driven compression scheme for embedded systems. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pp. 95–104, New York, NY, USA, 2006. ACM Press.

[Witt94] I. Witten, A. Moffat, and T. Bell. *Managing gigabytes: compressing and indexing documents and images*. Van Nostrand Reinhold, 1994.

[Wolf92] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pp. 81–91, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Xie03] Y. Xie, W. Wolf, and H. Lekatsas. Profile-driven selective code compression. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 1530–1591, Washington, DC, USA, 2003. IEEE Computer Society.

[Zhan06a] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI '06: Proceedings of the 2006 ACM*

SIGPLAN conference on Programming language design and implementation, pp. 169–180, New York, NY, USA, 2006. ACM Press.

[Zhan06b] Y. Zhang and R. Gupta. Compressing heap data for improved memory performance: Research Articles. *Software—Practice & Experience*, 36(10):1081–1111, 2006.

[Ziv77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.